Cloud-Init

Release 17.1

Sep 21, 2017

Contents

Everything about cloud-init, a set of python scripts and utilities to make your cloud images be all they can be!

CHAPTER 1

Summary

Cloud-init is the *defacto* multi-distribution package that handles early initialization of a cloud instance.

Capabilities

- Setting a default locale
- Setting an instance hostname
- Generating instance SSH private keys
- Adding SSH keys to a user's `.ssh/authorized_keys` so they can log in
- Setting up ephemeral mount points
- Configuring network devices

User configurability

Cloud-init 's behavior can be configured via user-data.

User-data can be given by the user at instance launch time.

This is done via the `--user-data` or `--user-data-file` argument to ec2-run-instances for example.

- Check your local clients documentation for how to provide a *user-data* string or *user-data* file for usage by cloud-init on instance creation.

Feature detection

Newer versions of cloud-init may have a list of additional features that they support. This allows other applications to detect what features the installed cloud-init supports without having to parse its version number. If present, this list of features will be located at `cloudinit.version.FEATURES`.

Currently defined feature names include:

- `NETWORK_CONFIG_V1` support for v1 networking configuration, see *Networking Config Version 1* documentation for examples.
- `NETWORK_CONFIG_V2` support for v2 networking configuration, see *Networking Config Version 2* documentation for examples.

CLI Interface :

`cloud-init features` will print out each feature supported. If cloud-init does not have the features subcommand, it also does not support any features described in this document.

```
% cloud-init --help
usage: cloud-init [-h] [--version] [--file FILES] [--debug] [--force]
                  {init,modules,query,single,dhclient-hook,features} ...

optional arguments:
  -h, --help            show this help message and exit
  --version, -v         show program's version number and exit
  --file FILES, -f FILES
                        additional yaml configuration files to use
  --debug, -d           show additional pre-action logging (default: False)
  --force               force running even if no datasource is found (use at
                        your own risk)

Subcommands:
  {init,modules,single,dhclient-hook,features,analyze,devel}
    init                initializes cloud-init and performs initial modules
    modules             activates modules using a given configuration key
    single              run a single module
    dhclient-hook       run the dhclient hookto record network info
    features            list defined features
    analyze             Devel tool: Analyze cloud-init logs and data
    devel               Run development tools

% cloud-init features
NETWORK_CONFIG_V1
NETWORK_CONFIG_V2
```

Availability

It is currently installed in the Ubuntu Cloud Images and also in the official Ubuntu images available on EC2, Azure, GCE and many other clouds.

Versions for other systems can be (or have been) created for the following distributions:

- Ubuntu
- Fedora
- Debian

- RHEL
- CentOS
- *and more...*

So ask your distribution provider where you can obtain an image with it built-in if one is not already available

Formats

User data that will be acted upon by cloud-init must be in one of the following types.

Gzip Compressed Content

Content found to be gzip compressed will be uncompressed. The uncompressed data will then be used as if it were not compressed. This is typically useful because user-data is limited to ~16384[1] bytes.

Mime Multi Part Archive

This list of rules is applied to each part of this multi-part file. Using a mime-multi part file, the user can specify more than one type of data.

For example, both a user data script and a cloud-config type could be specified.

Supported content-types:

- text/x-include-once-url
- text/x-include-url
- text/cloud-config-archive
- text/upstart-job
- text/cloud-config
- text/part-handler
- text/x-shellscript
- text/cloud-boothook

Helper script to generate mime messages

```
#!/usr/bin/python

import sys

from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText

if len(sys.argv) == 1:
    print("%s input-file:type ..." % (sys.argv[0]))
    sys.exit(1)
```

[1] See your cloud provider for applicable user-data size limitations...

```
combined_message = MIMEMultipart()
for i in sys.argv[1:]:
    (filename, format_type) = i.split(":", 1)
    with open(filename) as fh:
        contents = fh.read()
    sub_message = MIMEText(contents, format_type, sys.getdefaultencoding())
    sub_message.add_header('Content-Disposition', 'attachment; filename="%s"' %␣
↪(filename))
    combined_message.attach(sub_message)

print(combined_message)
```

User-Data Script

Typically used by those who just want to execute a shell script.

Begins with: `#!` or `Content-Type: text/x-shellscript` when using a MIME archive.

Example

```
$ cat myscript.sh

#!/bin/sh
echo "Hello World.  The time is now $(date -R)!" | tee /root/output.txt

$ euca-run-instances --key mykey --user-data-file myscript.sh ami-a07d95c9
```

Include File

This content is a `include` file.

The file contains a list of urls, one per line. Each of the URLs will be read, and their content will be passed through this same set of rules. Ie, the content read from the URL can be gzipped, mime-multi-part, or plain text. If an error occurs reading a file the remaining files will not be read.

Begins with: `#include` or `Content-Type: text/x-include-url` when using a MIME archive.

Cloud Config Data

Cloud-config is the simplest way to accomplish some things via user-data. Using cloud-config syntax, the user can specify certain things in a human friendly format.

These things include:

- apt upgrade should be run on first boot
- a different apt mirror should be used
- additional apt sources should be added
- certain ssh keys should be imported
- *and many more...*

Note: The file must be valid yaml syntax.

See the *Cloud config examples* section for a commented set of examples of supported cloud config formats.

Begins with: `#cloud-config` or `Content-Type: text/cloud-config` when using a MIME archive.

Upstart Job

Content is placed into a file in `/etc/init`, and will be consumed by upstart as any other upstart job.

Begins with: `#upstart-job` or `Content-Type: text/upstart-job` when using a MIME archive.

Cloud Boothook

This content is `boothook` data. It is stored in a file under `/var/lib/cloud` and then executed immediately. This is the earliest `hook` available. Note, that there is no mechanism provided for running only once. The boothook must take care of this itself. It is provided with the instance id in the environment variable `INSTANCE_I`. This could be made use of to provide a 'once-per-instance' type of functionality.

Begins with: `#cloud-boothook` or `Content-Type: text/cloud-boothook` when using a MIME archive.

Part Handler

This is a `part-handler`: It contains custom code for either supporting new mime-types in multi-part user data, or overriding the existing handlers for supported mime-types. It will be written to a file in `/var/lib/cloud/data` based on its filename (which is generated). This must be python code that contains a `list_types` function and a `handle_part` function. Once the section is read the `list_types` method will be called. It must return a list of mime-types that this part-handler handles. Because mime parts are processed in order, a `part-handler` part must precede any parts with mime-types it is expected to handle in the same user data.

The `handle_part` function must be defined like:

```
def handle_part(data, ctype, filename, payload):
  # data = the cloudinit object
  # ctype = "__begin__", "__end__", or the mime-type of the part that is being␣
↪handled.
  # filename = the filename of the part (or a generated filename if none is present␣
↪in mime data)
  # payload = the parts' content
```

Cloud-init will then call the `handle_part` function once before it handles any parts, once per part received, and once after all parts have been handled. The `'__begin__'` and `'__end__'` sentinels allow the part handler to do initialization or teardown before or after receiving any parts.

Begins with: `#part-handler` or `Content-Type: text/part-handler` when using a MIME archive.

Example

```
#part-handler
# vi: syntax=python ts=4

def list_types():
    # return a list of mime-types that are handled by this module
```

```
    return(["text/plain", "text/go-cubs-go"])

def handle_part(data,ctype,filename,payload):
    # data: the cloudinit object
    # ctype: '__begin__', '__end__', or the specific mime-type of the part
    # filename: the filename for the part, or dynamically generated part if
    #           no filename is given attribute is present
    # payload: the content of the part (empty for begin or end)
    if ctype == "__begin__":
       print "my handler is beginning"
       return
    if ctype == "__end__":
       print "my handler is ending"
       return

    print "==== received ctype=%s filename=%s ====" % (ctype,filename)
    print payload
    print "==== end ctype=%s filename=%s" % (ctype, filename)
```

Also this blog post offers another example for more advanced usage.

Directory layout

Cloudinits's directory structure is somewhat different from a regular application:

```
/var/lib/cloud/
    - data/
       - instance-id
       - previous-instance-id
       - datasource
       - previous-datasource
       - previous-hostname
    - handlers/
    - instance
    - instances/
        i-00000XYZ/
          - boot-finished
          - cloud-config.txt
          - datasource
          - handlers/
          - obj.pkl
          - scripts/
          - sem/
          - user-data.txt
          - user-data.txt.i
    - scripts/
       - per-boot/
       - per-instance/
       - per-once/
    - seed/
    - sem/
```

`/var/lib/cloud`

The main directory containing the cloud-init specific subdirectories. It is typically located at `/var/lib` but there are certain configuration scenarios where this can be altered.

TBD, describe this overriding more.

`data/`

Contains information related to instance ids, datasources and hostnames of the previous and current instance if they are different. These can be examined as needed to determine any information related to a previous boot (if applicable).

`handlers/`

Custom `part-handlers` code is written out here. Files that end up here are written out with in the scheme of `part-handler-XYZ` where `XYZ` is the handler number (the first handler found starts at 0).

`instance`

A symlink to the current `instances/` subdirectory that points to the currently active instance (which is active is dependent on the datasource loaded).

`instances/`

All instances that were created using this image end up with instance identifier subdirectories (and corresponding data for each instance). The currently active instance will be symlinked the `instance` symlink file defined previously.

`scripts/`

Scripts that are downloaded/created by the corresponding `part-handler` will end up in one of these subdirectories.

`seed/`

TBD

`sem/`

Cloud-init has a concept of a module semaphore, which basically consists of the module name and its frequency. These files are used to ensure a module is only ran *per-once*, *per-instance*, *per-always*. This folder contains semaphore *files* which are only supposed to run *per-once* (not tied to the instance id).

Cloud config examples

Including users and groups

```
# Add groups to the system
# The following example adds the ubuntu group with members foo and bar and
# the group cloud-users.
groups:
  - ubuntu: [foo,bar]
  - cloud-users

# Add users to the system. Users are added after groups are added.
users:
  - default
  - name: foobar
    gecos: Foo B. Bar
    primary-group: foobar
    groups: users
    selinux-user: staff_u
    expiredate: 2012-09-01
    ssh-import-id: foobar
```

```
    lock_passwd: false
    passwd: $6$j212wezy$7H/1LT4f9/
↪N3wpgNunhsIqtMj62OKiS3nyNwuizouQc3u7MbYCarYeAHWYPYb2FT.lbioDm2RrkJPb9BZMN1O/
  - name: barfoo
    gecos: Bar B. Foo
    sudo: ALL=(ALL) NOPASSWD:ALL
    groups: users, admin
    ssh-import-id: None
    lock_passwd: true
    ssh-authorized-keys:
      - <ssh pub key 1>
      - <ssh pub key 2>
  - name: cloudy
    gecos: Magic Cloud App Daemon User
    inactive: true
    system: true
  - snapuser: joe@joeuser.io

# Valid Values:
#   name: The user's login name
#   gecos: The user name's real name, i.e. "Bob B. Smith"
#   homedir: Optional. Set to the local path you want to use. Defaults to
#            /home/<username>
#   primary-group: define the primary group. Defaults to a new group created
#            named after the user.
#   groups:  Optional. Additional groups to add the user to. Defaults to none
#   selinux-user:  Optional. The SELinux user for the user's login, such as
#            "staff_u". When this is omitted the system will select the default
#            SELinux user.
#   lock_passwd: Defaults to true. Lock the password to disable password login
#   inactive: Create the user as inactive
#   passwd: The hash -- not the password itself -- of the password you want
#            to use for this user. You can generate a safe hash via:
#                mkpasswd --method=SHA-512 --rounds=4096
#            (the above command would create from stdin an SHA-512 password hash
#            with 4096 salt rounds)
#
#            Please note: while the use of a hashed password is better than
#                plain text, the use of this feature is not ideal. Also,
#                using a high number of salting rounds will help, but it should
#                not be relied upon.
#
#                To highlight this risk, running John the Ripper against the
#                example hash above, with a readily available wordlist, revealed
#                the true password in 12 seconds on a i7-2620QM.
#
#                In other words, this feature is a potential security risk and is
#                provided for your convenience only. If you do not fully trust the
#                medium over which your cloud-config will be transmitted, then you
#                should use SSH authentication only.
#
#                You have thus been warned.
#   no-create-home: When set to true, do not create home directory.
#   no-user-group: When set to true, do not create a group named after the user.
#   no-log-init: When set to true, do not initialize lastlog and faillog database.
#   ssh-import-id: Optional. Import SSH ids
#   ssh-authorized-keys: Optional. [list] Add keys to user's authorized keys file
#   sudo: Defaults to none. Set to the sudo string you want to use, i.e.
```

```
#            ALL=(ALL) NOPASSWD:ALL. To add multiple rules, use the following
#            format.
#                sudo:
#                    - ALL=(ALL) NOPASSWD:/bin/mysql
#                    - ALL=(ALL) ALL
#            Note: Please double check your syntax and make sure it is valid.
#                cloud-init does not parse/check the syntax of the sudo
#                directive.
#   system: Create the user as a system user. This means no home directory.
#   snapuser: Create a Snappy (Ubuntu-Core) user via the snap create-user
#             command available on Ubuntu systems.  If the user has an account
#             on the Ubuntu SSO, specifying the email will allow snap to
#             request a username and any public ssh keys and will import
#             these into the system with username specifed by SSO account.
#             If 'username' is not set in SSO, then username will be the
#             shortname before the email domain.
#

# Default user creation:
#
# Unless you define users, you will get a 'ubuntu' user on ubuntu systems with the
# legacy permission (no password sudo, locked user, etc). If however, you want
# to have the 'ubuntu' user in addition to other users, you need to instruct
# cloud-init that you also want the default user. To do this use the following
# syntax:
#    users:
#      - default
#      - bob
#      - ....
#  foobar: ...
#
# users[0] (the first user in users) overrides the user directive.
#
# The 'default' user above references the distro's config:
# system_info:
#   default_user:
#    name: Ubuntu
#    plain_text_passwd: 'ubuntu'
#    home: /home/ubuntu
#    shell: /bin/bash
#    lock_passwd: True
#    gecos: Ubuntu
#    groups: [adm, audio, cdrom, dialout, floppy, video, plugdev, dip, netdev]
```

Writing out arbitrary files

```
#cloud-config
# vim: syntax=yaml
#
# This is the configuration syntax that the write_files module
# will know how to understand. encoding can be given b64 or gzip or (gz+b64).
# The content will be decoded accordingly and then written to the path that is
# provided.
#
# Note: Content strings here are truncated for example purposes.
write_files:
```

```
-   encoding: b64
    content: CiMgVGhpcyBmaWxlIGNvbnRyb2xzIHRoZSBzdGF0ZSBvZiBTRUxpbnV4...
    owner: root:root
    path: /etc/sysconfig/selinux
    permissions: '0644'
-   content: |
        # My new /etc/sysconfig/samba file

        SMBDOPTIONS="-D"
    path: /etc/sysconfig/samba
-   content: !!binary |
        f0VMRgIBAQAAAAAAAAAAAAIAPgABAAAAwARAAAAAAABAAAAAAAAAAJAVAAAAAAAAAAAAAAEAAOAAI
        AEAAHgAdAAYAAAAFAAAAQAAAAAAAAABAAEAAAAAAAEAAQAAAAAAAwAEAAAAAAADAAQAAAAAAAAgA
        AAAAAAAAAwAAAAQAAAAAAgAAAAAAAAACQAAAAAAAAAJAAAAAAAAcAAAAAAAAABwAAAAAAAAAQAA
        ....
    path: /bin/arch
    permissions: '0555'
-   encoding: gzip
    content: !!binary |
        H4sIAIDb/U8C/1NW1E/KzNMvzuBKTc7IV8hIzcnJVyjPL8pJ4QIA6N+MVxsAAAA=
    path: /usr/bin/hello
    permissions: '0755'
```

Adding a yum repository

```
#cloud-config
# vim: syntax=yaml
#
# Add yum repository configuration to the system
#
# The following example adds the file /etc/yum.repos.d/epel_testing.repo
# which can then subsequently be used by yum for later operations.
yum_repos:
    # The name of the repository
    epel-testing:
        # Any repository configuration options
        # See: man yum.conf
        #
        # This one is required!
        baseurl: http://download.fedoraproject.org/pub/epel/testing/5/$basearch
        enabled: false
        failovermethod: priority
        gpgcheck: true
        gpgkey: file:///etc/pki/rpm-gpg/RPM-GPG-KEY-EPEL
        name: Extra Packages for Enterprise Linux 5 - Testing
```

Configure an instances trusted CA certificates

```
#cloud-config
#
# This is an example file to configure an instance's trusted CA certificates
# system-wide for SSL/TLS trust establishment when the instance boots for the
# first time.
#
```

```
# Make sure that this file is valid yaml before starting instances.
# It should be passed as user-data when starting the instance.

ca-certs:
  # If present and set to True, the 'remove-defaults' parameter will remove
  # all the default trusted CA certificates that are normally shipped with
  # Ubuntu.
  # This is mainly for paranoid admins - most users will not need this
  # functionality.
  remove-defaults: true

  # If present, the 'trusted' parameter should contain a certificate (or list
  # of certificates) to add to the system as trusted CA certificates.
  # Pay close attention to the YAML multiline list syntax.  The example shown
  # here is for a list of multiline certificates.
  trusted:
  - |
    -----BEGIN CERTIFICATE-----
    YOUR-ORGS-TRUSTED-CA-CERT-HERE
    -----END CERTIFICATE-----
  - |
    -----BEGIN CERTIFICATE-----
    YOUR-ORGS-TRUSTED-CA-CERT-HERE
    -----END CERTIFICATE-----
```

Configure an instances resolv.conf

Note: when using a config drive and a RHEL like system resolv.conf will also be managed 'automatically' due to the available information provided for dns servers in the config drive network format. For those that wish to have different settings use this module.

```
#cloud-config
#
# This is an example file to automatically configure resolv.conf when the
# instance boots for the first time.
#
# Ensure that your yaml is valid and pass this as user-data when starting
# the instance. Also be sure that your cloud.cfg file includes this
# configuration module in the appropriate section.
#
manage_resolv_conf: true

resolv_conf:
  nameservers: ['8.8.4.4', '8.8.8.8']
  searchdomains:
    - foo.example.com
    - bar.example.com
  domain: example.com
  options:
    rotate: true
    timeout: 1
```

Install and run chef recipes

```
#cloud-config
#
# This is an example file to automatically install chef-client and run a
# list of recipes when the instance boots for the first time.
# Make sure that this file is valid yaml before starting instances.
# It should be passed as user-data when starting the instance.
#
# This example assumes the instance is 16.04 (xenial)


# The default is to install from packages.

# Key from https://packages.chef.io/chef.asc
apt:
  source1:
    source: "deb http://packages.chef.io/repos/apt/stable $RELEASE main"
    key: |
      -----BEGIN PGP PUBLIC KEY BLOCK-----
      Version: GnuPG v1.4.12 (Darwin)
      Comment: GPGTools - http://gpgtools.org

      mQGiBEppC7QRBADfsOkZU6KZK+YmKw4wev5mjKJEkVGlus+NxW8wItX5sGa6kdUu
      twAyj7Yr92rF+ICFEP3gGU6+lGo0Nve7KxkN/1W7/m3G4zuk+ccIKmjp8KS3qn99
      dxy64vcji9jIllVa+XXOGIp0G8GEaj7mbkixL/bMeGfdMlv8Gf2XPpp9vwCgn/GC
      JKacfnw7MpLKUHOYSlb//JsEAJqao3ViNfav83jJKEkD8cf59Y8xKia5OpZqTK5W
      ShVnNWS3U5IVQk10ZDH97Qn/YrK387H4CyhLE9mxPXs/ul18ioiaars/q2MEKU2I
      XKfV21eMLO9LYd6Ny/Kqj8o5WQK2J6+NAhSwvthZcIEphcFignIuobP+B5wNFQpe
      DbKfA/0WvN2OwFeWRcmmd3Hz7nHTpcnSF+4QX6yHRF/5BgxkG6IqBIACQbzPn6Hm
      sMtm/SVf11izmDqSsQptCrOZILfLX/mE+YOl+CwWSHhl+YsFts1WOuh1EhQD26aO
      Z84HuHV5HFRWjDLw9LriltBVQcXbpfSrRP5bdr7Wh8vhqJTPjrQnT3BzY29kZSBQ
      YWNrYWdlcyA8cGFja2FnZXNAb3BzY29kZS5jb20+iGAEExECACAFAkppC7QCGwMG
      CwkIBwMCBBUCCAMEFgIDAQIeAQIXgAAKCRApQKupg++Caj8sAKCOXmdG36gWji/K
      +o+XtBfvdMnFYQCfTCEWxRy2BnzLoBBFCjDSK6sJqCu0IENIRUYgUGFja2FnZXMg
      PHBhY2thZ2VzQGNoZWYuaW8+iGIEExECACIFAlQwYFECGwMGCwkIBwMCBhUIAgkK
      CwQWAgMBAh4BAheAAAoJECl1Aq6mD74JqX94An26z99XOHWpLN8ahzm7cp13t4Xid
      AJ9wVcgoUBzvgg91lKfv/34cmemZn7kCDQRKaQu0EAgAg7ZLCVGVTmLqBM6njZEd
      Zbv+mZbvwLBSomdiqddE6u3eH0X3GuwaQfQWHUVG2yedyDMiG+EMtCdEeeRebTCz
      SNXQ8Xvi22hRPoEsBSwWLZI8/XNg0n0f1+GEr+mOKO0BxDB2DG7DA0nnEISxwFkK
      OFJFebR3fRsrWjj0KjDxkhse2ddU/jVz1BY7Nf8toZmwpBmdozETMOTx3LJy1HZ/
      Te9FJXJMUaB2lRyluv15MVWCKQJro4MQG/7QGcIfrIZNfAGJ32DDSjV7/YO+IpRY
      IL4CUBQ65suY4gYUG4jhRH6u7H1p99sdwsg5OIpBe/v2Vbc/tbwAB+eJJAp89Zeu
      twADBQf/ZcGoPhTGFuzbkcNRSIz+boaeWPoSxK2DyfScyCAuG41CY9+g0HIw9Sq8
      DuxQvJ+vrEJjNvNE3EAEdKl/zkXMZDb1EXjGwDi845TxEMhhD1dDw2qpHqnJ2mtE
      WpZ7juGwA3sGhi6FapO04tIGacCfNNHmlRGipyq5ZiKIRq9mLEndlECr8cwaKgkS
      0wWu+xmMZe7N5/t/TK19HXNh4tVacv0F3fYK54GUjt2FjCQV75USnmNY4KPTYLXA
      dzC364hEMlXpN21siIFgB04w+TXn5UF3B4FfAy5hevvr4DtV4MvMiGLu0oWjpaLC
      MpmrR3Ny2wkmO0h+vgri9uIP06ODWIhJBBgRAgAJBQJKaQu0AhsMAAoJECl1Aq6mD
      74Jq4hIAoJ5KrYS8kCwj26SAGzglwggpvt3CAJ0bekyky56vNqoegB+y4PQVDv4K
      zA==
      =IxPr
      -----END PGP PUBLIC KEY BLOCK-----

chef:

 # Valid values are 'gems' and 'packages' and 'omnibus'
 install_type: "packages"
```

```
 # Boolean: run 'install_type' code even if chef-client
 #          appears already installed.
 force_install: false

 # Chef settings
 server_url: "https://chef.yourorg.com"

 # Node Name
 # Defaults to the instance-id if not present
 node_name: "your-node-name"

 # Environment
 # Defaults to '_default' if not present
 environment: "production"

 # Default validation name is chef-validator
 validation_name: "yourorg-validator"
 # if validation_cert's value is "system" then it is expected
 # that the file already exists on the system.
 validation_cert: |
     -----BEGIN RSA PRIVATE KEY-----
     YOUR-ORGS-VALIDATION-KEY-HERE
     -----END RSA PRIVATE KEY-----

 # A run list for a first boot json, an example (not required)
 run_list:
  - "recipe[apache2]"
  - "role[db]"

 # Specify a list of initial attributes used by the cookbooks
 initial_attributes:
    apache:
      prefork:
        maxclients: 100
      keepalive: "off"

 # if install_type is 'omnibus', change the url to download
 omnibus_url: "https://www.chef.io/chef/install.sh"

 # if install_type is 'omnibus', pass pinned version string
 # to the install script
 omnibus_version: "12.3.0"


# Capture all subprocess output into a logfile
# Useful for troubleshooting cloud-init issues
output: {all: '| tee -a /var/log/cloud-init-output.log'}
```

Setup and run puppet

```
#cloud-config
#
# This is an example file to automatically setup and run puppetd
# when the instance boots for the first time.
# Make sure that this file is valid yaml before starting instances.
```

```
# It should be passed as user-data when starting the instance.
puppet:
 # Every key present in the conf object will be added to puppet.conf:
 # [name]
 # subkey=value
 #
 # For example the configuration below will have the following section
 # added to puppet.conf:
 # [puppetd]
 # server=puppetmaster.example.org
 # certname=i-0123456.ip-X-Y-Z.cloud.internal
 #
 # The puppmaster ca certificate will be available in
 # /var/lib/puppet/ssl/certs/ca.pem
 conf:
   agent:
     server: "puppetmaster.example.org"
     # certname supports substitutions at runtime:
     #   %i: instanceid
     #       Example: i-0123456
     #   %f: fqdn of the machine
     #       Example: ip-X-Y-Z.cloud.internal
     #
     # NB: the certname will automatically be lowercased as required by puppet
     certname: "%i.%f"
   # ca_cert is a special case. It won't be added to puppet.conf.
   # It holds the puppetmaster certificate in pem format.
   # It should be a multi-line string (using the | yaml notation for
   # multi-line strings).
   # The puppetmaster certificate is located in
   # /var/lib/puppet/ssl/ca/ca_crt.pem on the puppetmaster host.
   #
   ca_cert: |
     -----BEGIN CERTIFICATE-----
     MIICCTCCAXKgAwIBAgIBATANBgkqhkiG9w0BAQUFADANMQswCQYDVQQDDAJjYTAe
     Fw0xMDAyMTUxNzI5MjFaFw0xNTAyMTQxNzI5MjFaMA0xCzAJBgNVBAMMAmNhMIGf
     MA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCu7Q40sm47/E1Pf+r8AYb/V/FWGPgc
     b014OmNoX7dgCxTDvps/h8Vw555PdAFsW5+QhsGr31IJNI3kSYprFQcYf7A8tNWu
     1MASW2CfaEiOEi9F1R3R4Qlz4ix+iNoHiUDTjazw/tZwEdxaQXQVLwgTGRwVa+aA
     qbutJKi93MILLwIDAQABo3kwdzA4BglghkgBhvhCAQ0EKxYpUHVwcGV0IFJ1Ynkv
     T3BlblNTTCBHZW5lcmF0ZWQgQ2VydGlmaWNhdGUwDwYDVR0TAQH/BAUwAwEB/zAd
     BgNVHQ4EFgQUu4+jHB+GYE5Vxo+ol1OAhevspjAwCwYDVR0PBAQDAgEGMA0GCSqG
     SIb3DQEBBQUAA4GBAH/rxlUIjwNb3n7TXJcDJ6MMHUlwjr03BDJXKb34Ulndkpaf
     +GAlzPXWa7bO908M9I8RnPfvtKnteLbvgTK+h+zX1XCty+S2EQWk29i2AdoqOTxb
     hppiGMp0tT5Havu4aceCXiy2crVcudj3NFciy8X66SoECemW9UYDCb9T5D0d
     -----END CERTIFICATE-----
```

Add apt repositories

```
#cloud-config

# Add apt repositories
#
# Default: auto select based on cloud metadata
#  in ec2, the default is <region>.archive.ubuntu.com
# apt:
```

```
#   primary:
#     - arches [default]
#       uri:
#     use the provided mirror
#       search:
#     search the list for the first mirror.
#     this is currently very limited, only verifying that
#     the mirror is dns resolvable or an IP address
#
# if neither mirror is set (the default)
# then use the mirror provided by the DataSource found.
# In EC2, that means using <region>.ec2.archive.ubuntu.com
#
# if no mirror is provided by the DataSource, but 'search_dns' is
# true, then search for dns names '<distro>-mirror' in each of
# - fqdn of this host per cloud metadata
# - localdomain
# - no domain (which would search domains listed in /etc/resolv.conf)
# If there is a dns entry for <distro>-mirror, then it is assumed that there
# is a distro mirror at http://<distro>-mirror.<domain>/<distro>
#
# That gives the cloud provider the opportunity to set mirrors of a distro
# up and expose them only by creating dns entries.
#
# if none of that is found, then the default distro mirror is used
apt:
  primary:
    - arches: [default]
      uri: http://us.archive.ubuntu.com/ubuntu/
# or
apt:
  primary:
    - arches: [default]
      search:
        - http://local-mirror.mydomain
        - http://archive.ubuntu.com
# or
apt:
  primary:
    - arches: [default]
      search_dns: True
```

Run commands on first boot

```
#cloud-config

# boot commands
# default: none
# this is very similar to runcmd, but commands run very early
# in the boot process, only slightly after a 'boothook' would run.
# bootcmd should really only be used for things that could not be
# done later in the boot process.  bootcmd is very much like
# boothook, but possibly with more friendly.
# - bootcmd will run on every boot
# - the INSTANCE_ID variable will be set to the current instance id.
# - you can use 'cloud-init-per' command to help only run once
```

```
bootcmd:
 - echo 192.168.1.130 us.archive.ubuntu.com >> /etc/hosts
 - [ cloud-init-per, once, mymkfs, mkfs, /dev/vdb ]
```

```
#cloud-config

# run commands
# default: none
# runcmd contains a list of either lists or a string
# each item will be executed in order at rc.local like level with
# output to the console
# - runcmd only runs during the first boot
# - if the item is a list, the items will be properly executed as if
#   passed to execve(3) (with the first arg as the command).
# - if the item is a string, it will be simply written to the file and
#   will be interpreted by 'sh'
#
# Note, that the list has to be proper yaml, so you have to quote
# any characters yaml would eat (':' can be problematic)
runcmd:
 - [ ls, -l, / ]
 - [ sh, -xc, "echo $(date) ': hello world!'" ]
 - [ sh, -c, echo "=========hello world'=========" ]
 - ls -l /root
 - [ wget, "http://slashdot.org", -O, /tmp/index.html ]
```

Alter the completion message

```
#cloud-config

# final_message
# default: cloud-init boot finished at $TIMESTAMP. Up $UPTIME seconds
# this message is written by cloud-final when the system is finished
# its first boot
final_message: "The system is finally up, after $UPTIME seconds"
```

Install arbitrary packages

```
#cloud-config

# Install additional packages on first boot
#
# Default: none
#
# if packages are specified, this apt_update will be set to true
#
# packages may be supplied as a single package name or as a list
# with the format [<package>, <version>] wherein the specifc
# package version will be installed.
packages:
 - pwgen
 - pastebinit
 - [libpython2.7, 2.7.3-0ubuntu3.1]
```

Update apt database on first boot

```
#cloud-config
# Update apt database on first boot (run 'apt-get update').
# Note, if packages are given, or package_upgrade is true, then
# update will be done independent of this setting.
#
# Default: false
# Aliases: apt_update
package_update: false
```

Run apt or yum upgrade

```
#cloud-config

# Upgrade the instance on first boot
# (ie run apt-get upgrade)
#
# Default: false
# Aliases: apt_upgrade
package_upgrade: true
```

Adjust mount points mounted

```
#cloud-config

# set up mount points
# 'mounts' contains a list of lists
#  the inner list are entries for an /etc/fstab line
#  ie : [ fs_spec, fs_file, fs_vfstype, fs_mntops, fs-freq, fs_passno ]
#
# default:
# mounts:
#  - [ ephemeral0, /mnt ]
#  - [ swap, none, swap, sw, 0, 0 ]
#
# in order to remove a previously listed mount (ie, one from defaults)
# list only the fs_spec.  For example, to override the default, of
# mounting swap:
# - [ swap ]
# or
# - [ swap, null ]
#
# - if a device does not exist at the time, an entry will still be
#   written to /etc/fstab.
# - '/dev' can be ommitted for device names that begin with: xvd, sd, hd, vd
# - if an entry does not have all 6 fields, they will be filled in
#   with values from 'mount_default_fields' below.
#
# Note, that you should set 'nofail' (see man fstab) for volumes that may not
# be attached at instance boot (or reboot).
#
mounts:
 - [ ephemeral0, /mnt, auto, "defaults,noexec" ]
```

```
 - [ sdc, /opt/data ]
 - [ xvdh, /opt/data, "auto", "defaults,nofail", "0", "0" ]
 - [ dd, /dev/zero ]

# mount_default_fields
# These values are used to fill in any entries in 'mounts' that are not
# complete.  This must be an array, and must have 7 fields.
mount_default_fields: [ None, None, "auto", "defaults,nofail", "0", "2" ]


# swap can also be set up by the 'mounts' module
# default is to not create any swap files, because 'size' is set to 0
swap:
   filename: /swap.img
   size: "auto" # or size in bytes
   maxsize: size in bytes
```

Call a url when finished

```
#cloud-config

# phone_home: if this dictionary is present, then the phone_home
# cloud-config module will post specified data back to the given
# url
# default: none
# phone_home:
#  url: http://my.foo.bar/$INSTANCE/
#  post: all
#  tries: 10
#
phone_home:
 url: http://my.example.com/$INSTANCE_ID/
 post: [ pub_key_dsa, pub_key_rsa, pub_key_ecdsa, instance_id ]
```

Reboot/poweroff when finished

```
#cloud-config

## poweroff or reboot system after finished
# default: none
#
# power_state can be used to make the system shutdown, reboot or
# halt after boot is finished.  This same thing can be acheived by
# user-data scripts or by runcmd by simply invoking 'shutdown'.
#
# Doing it this way ensures that cloud-init is entirely finished with
# modules that would be executed, and avoids any error/log messages
# that may go to the console as a result of system services like
# syslog being taken down while cloud-init is running.
#
# If you delay '+5' (5 minutes) and have a timeout of
# 120 (2 minutes), then the max time until shutdown will be 7 minutes.
# cloud-init will invoke 'shutdown +5' after the process finishes, or
# when 'timeout' seconds have elapsed.
```

```
#
# delay: form accepted by shutdown.  default is 'now'. other format
#        accepted is +m (m in minutes)
# mode: required. must be one of 'poweroff', 'halt', 'reboot'
# message: provided as the message argument to 'shutdown'. default is none.
# timeout: the amount of time to give the cloud-init process to finish
#          before executing shutdown.
# condition: apply state change only if condition is met.
#            May be boolean True (always met), or False (never met),
#            or a command string or list to be executed.
#            command's exit code indicates:
#               0: condition met
#               1: condition not met
#            other exit codes will result in 'not met', but are reserved
#            for future use.
#
power_state:
 delay: "+30"
 mode: poweroff
 message: Bye Bye
 timeout: 30
 condition: True
```

Configure instances ssh-keys

```
#cloud-config

# add each entry to ~/.ssh/authorized_keys for the configured user or the
# first user defined in the user definition directive.
ssh_authorized_keys:
  - ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAGEA3FSyQwBI6Z+nCSjUUk8EEAnnkhXlukKoUPND/
↪RRClWz2s5TCzIkd3Ou5+Cyz71X0XmazM3l5WgeErvtIwQMyT1KjNoMhoJMrJnWqQPOt5Q8zWd9qG7PBl9+eiH5qV7NZ␣
↪mykey@host
  - ssh-rsa␣
↪AAAAB3NzaC1yc2EAAAABIwAAAQEA3I7VUf2l5gSn5uavROsc5HRDpZdQueUq5ozemNSj8T7enqKHOEaFoU2VoPgGEWC9RyzSQVe
↪+i1D+ey3ONkZLN+LQ714cgj8fRS4Hj29SCmXp5Kt5/82cD/VN3NtHw== smoser@brickies

# Send pre-generated ssh private keys to the server
# If these are present, they will be written to /etc/ssh and
# new random keys will not be generated
#  in addition to 'rsa' and 'dsa' as shown below, 'ecdsa' is also supported
ssh_keys:
  rsa_private: |
    -----BEGIN RSA PRIVATE KEY-----
    MIIBxwIBAAJhAKD0YSHy73nUgysO13XsJmd4fHiFyQ+00R7VVu2iV9Qcon2LZS/x
    1cydPZ4pQpfjEha6WxZ6o8ci/Ea/w0n+0HGPwaxlEG2Z9inNtj3pgFrYcRztfECb
    1j6HCibZbAzYtwIBIwJgO8h72WjcmvcpZ8OvHSvTwAguO2TkR6mPgHsgSaKy6GJo
    PUJnaZRWuba/HX0KGyhz19nPzLpzG5f0fYahlMJAyc13FV7K6kMBPXTRR6FxgHEg
    L0MPC7cdqAwOVNcPY6A7AjEA1bNaIjOzFN2sfZX0j7OMhQuc4zP7r80zaGc5oy6W
    p58hRAncFKEvnEq2CeL3vtuZAjEAwNBHpbNsBYTRPCHM7rZuG/iBtwp8Rxhc9I5w
    ixvzMgi+HpGLWzUIBS+P/XhekIjPAjA285rVmEP+DR255Ls65QbgYhJmTzIXQ2T9
    luLvcmFBC6l35Uc4gTgg4ALsmXLn71MCMGMpSWspEvuGInayTCL+vEjmNBT+FAdO
    W7D4zCpI43jRS9U06JVOeSc9CDk21wiA3wIwCTB/6uc8Cq85D9YqpM10FuHjKpnP
    REPPOyrAspdeOAV+6VKRavstea7+2DZmSUgE
    -----END RSA PRIVATE KEY-----

```

```
  rsa_public: ssh-rsa␣
↪AAAAB3NzaC1yc2EAAAABIwAAAGEAoPRhIfLvedSDKw7XdewmZ3h8eIXJD7TRHtVW7aJX1ByifYtlL/
↪HVzJ09nilCl+MSFrpbFnqjxyL8Rr/DSf7QcY/BrGUQbZn2Kc22PemAWthxHO18QJvWPocKJtlsDNi3␣
↪smoser@localhost

  dsa_private: |
    -----BEGIN DSA PRIVATE KEY-----
    MIIBuwIBAAKBgQDP2HLu7pTExL89USyM0264RCyWX/CMLmukxX0Jdbm29ax8FBJT
    pLrO8TIXVY5rPAJm1dTHnpuyJhOvU9G7M8tPUABtzSJh4GVSHlwaCfycwcpLv9TX
    DgWIpSj+6EiHCyaRlB1/CBp9RiaB+10QcFbm+lapuET+/Au6vSDp9IRtlQIVAIMR
    8KucvUYbOEI+yv+5LW9u3z/BAoGBAI0q6JP+JvJmwZFaeCMMVxXUbqiSko/P1lsa
    LNNBHZ5/8MOUIm8rB2FC6ziidfueJpqTMqeQmSAlEBCwnwreUnGfRrKoJpyPNENY
    d15MG6N5J+z81sEcHFeprryZ+D3Ge9VjPq3Tf3NhKKwCDQ0240aPezbnjPeFm4mH
    bYxxcZ9GAoGAXmLIFSQgiAPu459rCKxT46tHJtM0QfnNiEnQLbFluefZ/yiI4DI3
    8UzTCOXLhUA7ybmZha+D/csj15Y9/BNFuO7unzVhikCQV9DTeXX46pG4s1o23JKC
    /QaYWNMZ7kTRv+wWow9MhGiVdML4ZN4XnifuO5krqAybngIy66PMEoQCFEIsKKWv
    99iziAH0KBMVbxy03Trz
    -----END DSA PRIVATE KEY-----

  dsa_public: ssh-dss AAAAB3NzaC1kc3MAAACBAM/
↪Ycu7ulMTEvz1RLIzTbrhELJZf8Iwua6TFfQl1ubb1rHwUElOkus7xMhdVjms8AmbV1Meem7ImE69T0bszy09QAG3NImHgZVIeXH
↪JzByku/
↪1NcOBYilKP7oSIcLJpGUHX8IGn1GJoH7XRBwVub6Vqm4RP78C7q9IOn0hG2VAAAAFQCDEfCrnL1GGzhCPsr/
↪uS1vbt8/wQAAAIEAjSrok/4m8mbBkVp4IwxXFdRuqJKSj8/WWxos00Ednn/
↪ww5QibysHYULrOKJ1+54mmpMyp5CZICUQELCfCt5ScZ9GsqgmnI80Q1h3Xkwbo3kn7PzWwRwcV6muvJn4PcZ71WM+rdN/
↪c2EorAINDTbjRo97NueM94WbiYdtjHFxn0YAAACAXmLIFSQgiAPu459rCKxT46tHJtM0QfnNiEnQLbFluefZ/
↪yiI4DI38UzTCOXLhUA7ybmZha+D/csj15Y9/BNFuO7unzVhikCQV9DTeXX46pG4s1o23JKC/
↪QaYWNMZ7kTRv+wWow9MhGiVdML4ZN4XnifuO5krqAybngIy66PMEoQ= smoser@localhost
```

Additional apt configuration

```
# apt_pipelining (configure Acquire::http::Pipeline-Depth)
# Default: disables HTTP pipelining. Certain web servers, such
# as S3 do not pipeline properly (LP: #948461).
# Valid options:
#   False/default: Disables pipelining for APT
#   None/Unchanged: Use OS default
#   Number: Set pipelining to some number (not recommended)
apt_pipelining: False

## apt config via system_info:
# under the 'system_info', you can customize cloud-init's interaction
# with apt.
#  system_info:
#    apt_get_command: [command, argument, argument]
#    apt_get_upgrade_subcommand: dist-upgrade
#
# apt_get_command:
#  To specify a different 'apt-get' command, set 'apt_get_command'.
#  This must be a list, and the subcommand (update, upgrade) is appended to it.
#  default is:
#    ['apt-get', '--option=Dpkg::Options::=--force-confold',
#     '--option=Dpkg::options::=--force-unsafe-io', '--assume-yes', '--quiet']
#
# apt_get_upgrade_subcommand: "dist-upgrade"
#  Specify a different subcommand for 'upgrade. The default is 'dist-upgrade'.
```

```
#  This is the subcommand that is invoked for package_upgrade.
#
# apt_get_wrapper:
#   command: eatmydata
#   enabled: [True, False, "auto"]
#

# Install additional packages on first boot
#
# Default: none
#
# if packages are specified, this apt_update will be set to true

packages: ['pastebinit']

apt:
  # The apt config consists of two major "areas".
  #
  # On one hand there is the global configuration for the apt feature.
  #
  # On one hand (down in this file) there is the source dictionary which allows
  # to define various entries to be considered by apt.

  ############################################################################
  # Section 1: global apt configuration
  #
  # The following examples number the top keys to ease identification in
  # discussions.

  # 1.1 preserve_sources_list
  #
  # Preserves the existing /etc/apt/sources.list
  # Default: false - do overwrite sources_list. If set to true then any
  # "mirrors" configuration will have no effect.
  # Set to true to avoid affecting sources.list. In that case only
  # "extra" source specifications will be written into
  # /etc/apt/sources.list.d/*
  preserve_sources_list: true

  # 1.2 disable_suites
  #
  # This is an empty list by default, so nothing is disabled.
  #
  # If given, those suites are removed from sources.list after all other
  # modifications have been made.
  # Suites are even disabled if no other modification was made,
  # but not if is preserve_sources_list is active.
  # There is a special alias "$RELEASE" as in the sources that will be replace
  # by the matching release.
  #
  # To ease configuration and improve readability the following common ubuntu
  # suites will be automatically mapped to their full definition.
  # updates   => $RELEASE-updates
  # backports => $RELEASE-backports
  # security  => $RELEASE-security
  # proposed  => $RELEASE-proposed
  # release   => $RELEASE
  #
```

```
  # There is no harm in specifying a suite to be disabled that is not found in
  # the source.list file (just a no-op then)
  #
  # Note: Lines don't get deleted, but disabled by being converted to a comment.
  # The following example disables all usual defaults except $RELEASE-security.
  # On top it disables a custom suite called "mysuite"
  disable_suites: [$RELEASE-updates, backports, $RELEASE, mysuite]

  # 1.3 primary/security archives
  #
  # Default: none - instead it is auto select based on cloud metadata
  # so if neither "uri" nor "search", nor "search_dns" is set (the default)
  # then use the mirror provided by the DataSource found.
  # In EC2, that means using <region>.ec2.archive.ubuntu.com
  #
  # define a custom (e.g. localized) mirror that will be used in sources.list
  # and any custom sources entries for deb / deb-src lines.
  #
  # One can set primary and security mirror to different uri's
  # the child elements to the keys primary and secondary are equivalent
  primary:
    # arches is list of architectures the following config applies to
    # the special keyword "default" applies to any architecture not explicitly
    # listed.
    - arches: [amd64, i386, default]
      # uri is just defining the target as-is
      uri: http://us.archive.ubuntu.com/ubuntu
      #
      # via search one can define lists that are tried one by one.
      # The first with a working DNS resolution (or if it is an IP) will be
      # picked. That way one can keep one configuration for multiple
      # subenvironments that select the working one.
      search:
        - http://cool.but-sometimes-unreachable.com/ubuntu
        - http://us.archive.ubuntu.com/ubuntu
      # if no mirror is provided by uri or search but 'search_dns' is
      # true, then search for dns names '<distro>-mirror' in each of
      # - fqdn of this host per cloud metadata
      # - localdomain
      # - no domain (which would search domains listed in /etc/resolv.conf)
      # If there is a dns entry for <distro>-mirror, then it is assumed that
      # there is a distro mirror at http://<distro>-mirror.<domain>/<distro>
      #
      # That gives the cloud provider the opportunity to set mirrors of a distro
      # up and expose them only by creating dns entries.
      #
      # if none of that is found, then the default distro mirror is used
      search_dns: true
      #
      # If multiple of a category are given
      #   1. uri
      #   2. search
      #   3. search_dns
      # the first defining a valid mirror wins (in the order as defined here,
      # not the order as listed in the config).
      #
    - arches: [s390x, arm64]
      # as above, allowing to have one config for different per arch mirrors
```

```
  # security is optional, if not defined it is set to the same value as primary
  security:
      uri: http://security.ubuntu.com/ubuntu
  # If search_dns is set for security the searched pattern is:
  #   <distro>-security-mirror

  # if no mirrors are specified at all, or all lookups fail it will try
  # to get them from the cloud datasource and if those neither provide one fall
  # back to:
  #   primary: http://archive.ubuntu.com/ubuntu
  #   security: http://security.ubuntu.com/ubuntu

  # 1.4 sources_list
  #
  # Provide a custom template for rendering sources.list
  # without one provided cloud-init uses builtin templates for
  # ubuntu and debian.
  # Within these sources.list templates you can use the following replacement
  # variables (all have sane Ubuntu defaults, but mirrors can be overwritten
  # as needed (see above)):
  # => $RELEASE, $MIRROR, $PRIMARY, $SECURITY
  sources_list: | # written by cloud-init custom template
    deb $MIRROR $RELEASE main restricted
    deb-src $MIRROR $RELEASE main restricted
    deb $PRIMARY $RELEASE universe restricted
    deb $SECURITY $RELEASE-security multiverse

  # 1.5 conf
  #
  # Any apt config string that will be made available to apt
  # see the APT.CONF(5) man page for details what can be specified
  conf: | # APT config
    APT {
      Get {
        Assume-Yes "true";
        Fix-Broken "true";
      };
    };

  # 1.6 (http_|ftp_|https_)proxy
  #
  # Proxies are the most common apt.conf option, so that for simplified use
  # there is a shortcut for those. Those get automatically translated into the
  # correct Acquire::*::Proxy statements.
  #
  # note: proxy actually being a short synonym to http_proxy
  proxy: http://[[user][:pass]@]host[:port]/
  http_proxy: http://[[user][:pass]@]host[:port]/
  ftp_proxy: ftp://[[user][:pass]@]host[:port]/
  https_proxy: https://[[user][:pass]@]host[:port]/

  # 1.7 add_apt_repo_match
  #
  # 'source' entries in apt-sources that match this python regex
  # expression will be passed to add-apt-repository
  # The following example is also the builtin default if nothing is specified
  add_apt_repo_match: '^[\w-]+:\w'

```

```
  ##############################################################################
  # Section 2: source list entries
  #
  # This is a dictionary (unlike most block/net which are lists)
  #
  # The key of each source entry is the filename and will be prepended by
  # /etc/apt/sources.list.d/ if it doesn't start with a '/'.
  # If it doesn't end with .list it will be appended so that apt picks up it's
  # configuration.
  #
  # Whenever there is no content to be written into such a file, the key is
  # not used as filename - yet it can still be used as index for merging
  # configuration.
  #
  # The values inside the entries consost of the following optional entries:
  #   'source': a sources.list entry (some variable replacements apply)
  #   'keyid': providing a key to import via shortid or fingerprint
  #   'key': providing a raw PGP key
  #   'keyserver': specify an alternate keyserver to pull keys from that
  #                were specified by keyid

  # This allows merging between multiple input files than a list like:
  # cloud-config1
  # sources:
  #    s1: {'key': 'key1', 'source': 'source1'}
  # cloud-config2
  # sources:
  #    s2: {'key': 'key2'}
  #    s1: {'keyserver': 'foo'}
  # This would be merged to
  # sources:
  #    s1:
  #        keyserver: foo
  #        key: key1
  #        source: source1
  #    s2:
  #        key: key2
  #
  # The following examples number the subfeatures per sources entry to ease
  # identification in discussions.


  sources:
    curtin-dev-ppa.list:
      # 2.1 source
      #
      # Creates a file in /etc/apt/sources.list.d/ for the sources list entry
      # based on the key: "/etc/apt/sources.list.d/curtin-dev-ppa.list"
      source: "deb http://ppa.launchpad.net/curtin-dev/test-archive/ubuntu xenial main
↪"

      # 2.2 keyid
      #
      # Importing a gpg key for a given key id. Used keyserver defaults to
      # keyserver.ubuntu.com
      keyid: F430BBA5 # GPG key ID published on a key server

```

```
    ignored1:
      # 2.3 PPA shortcut
      #
      # Setup correct apt sources.list line and Auto-Import the signing key
      # from LP
      #
      # See https://help.launchpad.net/Packaging/PPA for more information
      # this requires 'add-apt-repository'. This will create a file in
      # /etc/apt/sources.list.d automatically, therefore the key here is
      # ignored as filename in those cases.
      source: "ppa:curtin-dev/test-archive"    # Quote the string

    my-repo2.list:
      # 2.4 replacement variables
      #
      # sources can use $MIRROR, $PRIMARY, $SECURITY and $RELEASE replacement
      # variables.
      # They will be replaced with the default or specified mirrors and the
      # running release.
      # The entry below would be possibly turned into:
      #   source: deb http://archive.ubuntu.com/ubuntu xenial multiverse
      source: deb $MIRROR $RELEASE multiverse

    my-repo3.list:
      # this would have the same end effect as 'ppa:curtin-dev/test-archive'
      source: "deb http://ppa.launchpad.net/curtin-dev/test-archive/ubuntu xenial main
↪"
      keyid: F430BBA5 # GPG key ID published on the key server
      filename: curtin-dev-ppa.list

    ignored2:
      # 2.5 key only
      #
      # this would only import the key without adding a ppa or other source spec
      # since this doesn't generate a source.list file the filename key is ignored
      keyid: F430BBA5 # GPG key ID published on a key server

    ignored3:
      # 2.6 key id alternatives
      #
      # Keyid's can also be specified via their long fingerprints
      keyid: B59D 5F15 97A5 04B7 E230  6DCA 0620 BBCF 0368 3F77

    ignored4:
      # 2.7 alternative keyservers
      #
      # One can also specify alternative keyservers to fetch keys from.
      keyid: B59D 5F15 97A5 04B7 E230  6DCA 0620 BBCF 0368 3F77
      keyserver: pgp.mit.edu


    my-repo4.list:
      # 2.8 raw key
      #
      # The apt signing key can also be specified by providing a pgp public key
      # block. Providing the PGP key this way is the most robust method for
      # specifying a key, as it removes dependency on a remote key server.
      #
```

```
      # As with keyid's this can be specified with or without some actual source
      # content.
      key: | # The value needs to start with -----BEGIN PGP PUBLIC KEY BLOCK-----
         -----BEGIN PGP PUBLIC KEY BLOCK-----
         Version: SKS 1.0.10

         mI0ESpA3UQEEALdZKVIMq0j6qWAXAyxSlF63SvPVIgxHPb9Nk0DZUixn+akqytxG4zKCONz6
         qLjoBBfHnynyVLfT4ihg9an1PqxRnTO+JKQxl8NgKGz6Pon569GtAOdWNKw15XKinJTDLjnj
         9y96ljJqRcpV9t/WsIcdJPcKFR5voHTEoABE2aEXABEBAAG0GUxhdW5jaHBhZCBQUEEgZm9y
         IEFsZXN0aWOItgQTAQIAIAUCSpA3UQIbAwYLCQgHAwIEFQIIAwQWAgMBAh4BAheAAAoJEA7H
         5Qi+CcVxWZ8D/1MyYvfj3FJPZUm2Yo1zZsQ657vHI9+pPouqflWOayRR9jbiyUFIn0VdQBrP
         t0FwvnOFArUovUWoKAEdqR8hPy3M3APUZjl5K4cMZR/xaMQeQRZ5CHpS4DBKURKAHC0ltS5o
         uBJKQOZm5iltJp15cgyIkBkGe8Mx18VFyVglAZey
         =Y2oI
         -----END PGP PUBLIC KEY BLOCK-----
```

Disk setup

```
# Cloud-init supports the creation of simple partition tables and file systems
# on devices.

# Default disk definitions for AWS
# --------------------------------
# (Not implemented yet, but provided for future documentation)

disk_setup:
   ephmeral0:
       table_type: 'mbr'
       layout: True
       overwrite: False

fs_setup:
   - label: None,
     filesystem: ext3
     device: ephemeral0
     partition: auto

# Default disk definitions for Windows Azure
# ------------------------------------------

device_aliases: {'ephemeral0': '/dev/sdb'}
disk_setup:
    ephemeral0:
         table_type: mbr
         layout: True
         overwrite: False

fs_setup:
    - label: ephemeral0
      filesystem: ext4
      device: ephemeral0.1
      replace_fs: ntfs


# Default disk definitions for SmartOS
# ------------------------------------
```

```
device_aliases: {'ephemeral0': '/dev/sdb'}
disk_setup:
    ephemeral0:
         table_type: mbr
         layout: False
         overwrite: False

fs_setup:
    - label: ephemeral0
      filesystem: ext3
      device: ephemeral0.0

# Cavaut for SmartOS: if ephemeral disk is not defined, then the disk will
#    not be automatically added to the mounts.


# The default definition is used to make sure that the ephemeral storage is
# setup properly.

# "disk_setup": disk partitioning
# --------------------------------

# The disk_setup directive instructs Cloud-init to partition a disk. The format is:

disk_setup:
   ephmeral0:
       table_type: 'mbr'
       layout: 'auto'
   /dev/xvdh:
       table_type: 'mbr'
       layout:
           - 33
           - [33, 82]
           - 33
       overwrite: True

# The format is a list of dicts of dicts. The first value is the name of the
# device and the subsequent values define how to create and layout the
# partition.
# The general format is:
#    disk_setup:
#        <DEVICE>:
#            table_type: 'mbr'
#            layout: <LAYOUT|BOOL>
#            overwrite: <BOOL>
#
# Where:
#    <DEVICE>: The name of the device. 'ephemeralX' and 'swap' are special
#                values which are specific to the cloud. For these devices
#                Cloud-init will look up what the real devices is and then
#                use it.
#
#                For other devices, the kernel device name is used. At this
#                time only simply kernel devices are supported, meaning
#                that device mapper and other targets may not work.
#
#                Note: At this time, there is no handling or setup of
```

```
#                 device mapper targets.
#
#    table_type=<TYPE>: Currently the following are supported:
#                     'mbr': default and setups a MS-DOS partition table
#                     'gpt': setups a GPT partition table
#
#                 Note: At this time only 'mbr' and 'gpt' partition tables
#                     are allowed. It is anticipated in the future that
#                     we'll also have "RAID" to create a mdadm RAID.
#
#    layout={...}: The device layout. This is a list of values, with the
#                 percentage of disk that partition will take.
#                 Valid options are:
#                     [<SIZE>, [<SIZE>, <PART_TYPE]]
#
#                 Where <SIZE> is the _percentage_ of the disk to use, while
#                 <PART_TYPE> is the numerical value of the partition type.
#
#                 The following setups two partitions, with the first
#                 partition having a swap label, taking 1/3 of the disk space
#                 and the remainder being used as the second partition.
#                     /dev/xvdh':
#                         table_type: 'mbr'
#                         layout:
#                             - [33,82]
#                             - 66
#                         overwrite: True
#
#                 When layout is "true" it means single partition the entire
#                 device.
#
#                 When layout is "false" it means don't partition or ignore
#                 existing partitioning.
#
#                 If layout is set to "true" and overwrite is set to "false",
#                 it will skip partitioning the device without a failure.
#
#    overwrite=<BOOL>: This describes whether to ride with saftey's on and
#                 everything holstered.
#
#                 'false' is the default, which means that:
#                     1. The device will be checked for a partition table
#                     2. The device will be checked for a file system
#                     3. If either a partition of file system is found, then
#                         the operation will be _skipped_.
#
#                 'true' is cowboy mode. There are no checks and things are
#                     done blindly. USE with caution, you can do things you
#                     really, really don't want to do.
#
#
# fs_setup: Setup the file system
# ------------------------------
#
# fs_setup describes the how the file systems are supposed to look.

fs_setup:
   - label: ephemeral0
```

```
    filesystem: 'ext3'
    device: 'ephemeral0'
    partition: 'auto'
  - label: mylabl2
    filesystem: 'ext4'
    device: '/dev/xvda1'
  - cmd: mkfs -t %(filesystem)s -L %(label)s %(device)s
    label: mylabl3
    filesystem: 'btrfs'
    device: '/dev/xvdh'

# The general format is:
#    fs_setup:
#        - label: <LABEL>
#          filesystem: <FS_TYPE>
#          device: <DEVICE>
#          partition: <PART_VALUE>
#          overwrite: <OVERWRITE>
#          replace_fs: <FS_TYPE>
#
# Where:
#     <LABEL>: The file system label to be used. If set to None, no label is
#        used.
#
#    <FS_TYPE>: The file system type. It is assumed that the there
#        will be a "mkfs.<FS_TYPE>" that behaves likes "mkfs". On a standard
#        Ubuntu Cloud Image, this means that you have the option of ext{2,3,4},
#        and vfat by default.
#
#    <DEVICE>: The device name. Special names of 'ephemeralX' or 'swap'
#        are allowed and the actual device is acquired from the cloud datasource.
#        When using 'ephemeralX' (i.e. ephemeral0), make sure to leave the
#        label as 'ephemeralX' otherwise there may be issues with the mounting
#        of the ephemeral storage layer.
#
#        If you define the device as 'ephemeralX.Y' then Y will be interpetted
#        as a partition value. However, ephermalX.0 is the _same_ as ephemeralX.
#
#    <PART_VALUE>:
#        Partition definitions are overwriten if you use the '<DEVICE>.Y' notation.
#
#        The valid options are:
#        "auto|any": tell cloud-init not to care whether there is a partition
#            or not. Auto will use the first partition that does not contain a
#            file system already. In the absence of a partition table, it will
#            put it directly on the disk.
#
#            "auto": If a file system that matches the specification in terms of
#            label, type and device, then cloud-init will skip the creation of
#            the file system.
#
#            "any": If a file system that matches the file system type and device,
#            then cloud-init will skip the creation of the file system.
#
#            Devices are selected based on first-detected, starting with partitions
#            and then the raw disk. Consider the following:
#                NAME     FSTYPE LABEL
#                xvdb
```

```
#                 |-xvdb1  ext4
#                 |-xvdb2
#                 |-xvdb3  btrfs  test
#                 \-xvdb4  ext4   test
#
#             If you ask for 'auto', label of 'test, and file system of 'ext4'
#             then cloud-init will select the 2nd partition, even though there
#             is a partition match at the 4th partition.
#
#             If you ask for 'any' and a label of 'test', then cloud-init will
#             select the 1st partition.
#
#             If you ask for 'auto' and don't define label, then cloud-init will
#             select the 1st partition.
#
#             In general, if you have a specific partition configuration in mind,
#             you should define either the device or the partition number. 'auto'
#             and 'any' are specifically intended for formating ephemeral storage or
#             for simple schemes.
#
#         "none": Put the file system directly on the device.
#
#         <NUM>: where NUM is the actual partition number.
#
#     <OVERWRITE>: Defines whether or not to overwrite any existing
#         filesystem.
#
#         "true": Indiscriminately destroy any pre-existing file system. Use at
#             your own peril.
#
#         "false": If an existing file system exists, skip the creation.
#
#     <REPLACE_FS>: This is a special directive, used for Windows Azure that
#         instructs cloud-init to replace a file system of <FS_TYPE>. NOTE:
#         unless you define a label, this requires the use of the 'any' partition
#         directive.
#
# Behavior Caveat: The default behavior is to _check_ if the file system exists.
#     If a file system matches the specification, then the operation is a no-op.
```

Register RedHat Subscription

```
#cloud-config

# register your Red Hat Enterprise Linux based operating system
#
# this cloud-init plugin is capable of registering by username
# and password *or* activation and org.  Following a successfully
# registration you can:
#    - auto-attach subscriptions
#    - set the service level
#    - add subscriptions based on its pool ID
#    - enable yum repositories based on its repo id
#    - disable yum repositories based on its repo id
#    - alter the rhsm_baseurl and server-hostname in the
#      /etc/rhsm/rhs.conf file
```

```

rh_subscription:
    username: joe@foo.bar

    ## Quote your password if it has symbols to be safe
    password: '1234abcd'

    ## If you prefer, you can use the activation key and
    ## org instead of username and password. Be sure to
    ## comment out username and password

    #activation-key: foobar
    #org: 12345

    ## Uncomment to auto-attach subscriptions to your system
    #auto-attach: True

    ## Uncomment to set the service level for your
    ##   subscriptions
    #service-level: self-support

    ## Uncomment to add pools (needs to be a list of IDs)
    #add-pool: []

    ## Uncomment to add or remove yum repos
    ##   (needs to be a list of repo IDs)
    #enable-repo: []
    #disable-repo: []

    ## Uncomment to alter the baseurl in /etc/rhsm/rhsm.conf
    #rhsm-baseurl: http://url

    ## Uncomment to alter the server hostname in
    ##  /etc/rhsm/rhsm.conf
    #server-hostname: foo.bar.com
```

Configure data sources

```
# Documentation on data sources configuration options
datasource:
  # Ec2
  Ec2:
    # timeout: the timeout value for a request at metadata service
    timeout : 50
    # The length in seconds to wait before giving up on the metadata
    # service.  The actual total wait could be up to
    #   len(resolvable_metadata_urls)*timeout
    max_wait : 120

    #metadata_url: a list of URLs to check for metadata services
    metadata_urls:
     - http://169.254.169.254:80
     - http://instance-data:8773

  MAAS:
    timeout : 50
```

```
    max_wait : 120

    # there are no default values for metadata_url or oauth credentials
    # If no credentials are present, non-authed attempts will be made.
    metadata_url: http://mass-host.localdomain/source
    consumer_key: Xh234sdkljf
    token_key: kjfhgb3n
    token_secret: 24uysdfx1w4

  NoCloud:
    # default seedfrom is None
    # if found, then it should contain a url with:
    #    <url>/user-data and <url>/meta-data
    # seedfrom: http://my.example.com/i-abcde
    seedfrom: None

    # fs_label: the label on filesystems to be searched for NoCloud source
    fs_label: cidata

    # these are optional, but allow you to basically provide a datasource
    # right here
    user-data: |
       # This is the user-data verbatum
    meta-data:
       instance-id: i-87018aed
       local-hostname: myhost.internal

  Azure:
    agent_command: [service, walinuxagent, start]
    set_hostname: True
    hostname_bounce:
      interface: eth0
      policy: on # [can be 'on', 'off' or 'force']

  SmartOS:
    # For KVM guests:
    # Smart OS datasource works over a serial console interacting with
    # a server on the other end. By default, the second serial console is the
    # device. SmartOS also uses a serial timeout of 60 seconds.
    serial_device: /dev/ttyS1
    serial_timeout: 60

    # For LX-Brand Zones guests:
    # Smart OS datasource works over a socket interacting with
    # the host on the other end. By default, the socket file is in
    # the native .zoncontrol directory.
    metadata_sockfile: /native/.zonecontrol/metadata.sock

    # a list of keys that will not be base64 decoded even if base64_all
    no_base64_decode: ['root_authorized_keys', 'motd_sys_info',
                       'iptables_disable']
    # a plaintext, comma delimited list of keys whose values are b64 encoded
    base64_keys: []
    # a boolean indicating that all keys not in 'no_base64_decode' are encoded
    base64_all: False
```

Create partitions and filesystems

```
# Cloud-init supports the creation of simple partition tables and file systems
# on devices.

# Default disk definitions for AWS
# --------------------------------
# (Not implemented yet, but provided for future documentation)

disk_setup:
   ephmeral0:
       table_type: 'mbr'
       layout: True
       overwrite: False

fs_setup:
   - label: None,
     filesystem: ext3
     device: ephemeral0
     partition: auto

# Default disk definitions for Windows Azure
# ------------------------------------------

device_aliases: {'ephemeral0': '/dev/sdb'}
disk_setup:
    ephemeral0:
         table_type: mbr
         layout: True
         overwrite: False

fs_setup:
    - label: ephemeral0
      filesystem: ext4
      device: ephemeral0.1
      replace_fs: ntfs


# Default disk definitions for SmartOS
# ------------------------------------

device_aliases: {'ephemeral0': '/dev/sdb'}
disk_setup:
    ephemeral0:
         table_type: mbr
         layout: False
         overwrite: False

fs_setup:
    - label: ephemeral0
      filesystem: ext3
      device: ephemeral0.0

# Cavaut for SmartOS: if ephemeral disk is not defined, then the disk will
#    not be automatically added to the mounts.


# The default definition is used to make sure that the ephemeral storage is
```

```
# setup properly.

# "disk_setup": disk partitioning
# --------------------------------

# The disk_setup directive instructs Cloud-init to partition a disk. The format is:

disk_setup:
   ephmeral0:
       table_type: 'mbr'
       layout: 'auto'
   /dev/xvdh:
       table_type: 'mbr'
       layout:
           - 33
           - [33, 82]
           - 33
       overwrite: True

# The format is a list of dicts of dicts. The first value is the name of the
# device and the subsequent values define how to create and layout the
# partition.
# The general format is:
#    disk_setup:
#        <DEVICE>:
#            table_type: 'mbr'
#            layout: <LAYOUT|BOOL>
#            overwrite: <BOOL>
#
# Where:
#    <DEVICE>: The name of the device. 'ephemeralX' and 'swap' are special
#                values which are specific to the cloud. For these devices
#                Cloud-init will look up what the real devices is and then
#                use it.
#
#                For other devices, the kernel device name is used. At this
#                time only simply kernel devices are supported, meaning
#                that device mapper and other targets may not work.
#
#                Note: At this time, there is no handling or setup of
#                device mapper targets.
#
#    table_type=<TYPE>: Currently the following are supported:
#                    'mbr': default and setups a MS-DOS partition table
#                    'gpt': setups a GPT partition table
#
#                Note: At this time only 'mbr' and 'gpt' partition tables
#                    are allowed. It is anticipated in the future that
#                    we'll also have "RAID" to create a mdadm RAID.
#
#    layout={...}: The device layout. This is a list of values, with the
#                percentage of disk that partition will take.
#                Valid options are:
#                    [<SIZE>, [<SIZE>, <PART_TYPE]]
#
#                Where <SIZE> is the _percentage_ of the disk to use, while
#                <PART_TYPE> is the numerical value of the partition type.
#
```

```
#                 The following setups two partitions, with the first
#                 partition having a swap label, taking 1/3 of the disk space
#                 and the remainder being used as the second partition.
#                     /dev/xvdh':
#                         table_type: 'mbr'
#                         layout:
#                             - [33,82]
#                             - 66
#                         overwrite: True
#
#                 When layout is "true" it means single partition the entire
#                 device.
#
#                 When layout is "false" it means don't partition or ignore
#                 existing partitioning.
#
#                 If layout is set to "true" and overwrite is set to "false",
#                 it will skip partitioning the device without a failure.
#
#    overwrite=<BOOL>: This describes whether to ride with saftey's on and
#                 everything holstered.
#
#                 'false' is the default, which means that:
#                     1. The device will be checked for a partition table
#                     2. The device will be checked for a file system
#                     3. If either a partition of file system is found, then
#                         the operation will be _skipped_.
#
#                 'true' is cowboy mode. There are no checks and things are
#                     done blindly. USE with caution, you can do things you
#                     really, really don't want to do.
#
#
# fs_setup: Setup the file system
# ------------------------------
#
# fs_setup describes the how the file systems are supposed to look.

fs_setup:
   - label: ephemeral0
     filesystem: 'ext3'
     device: 'ephemeral0'
     partition: 'auto'
   - label: mylabl2
     filesystem: 'ext4'
     device: '/dev/xvda1'
   - cmd: mkfs -t %(filesystem)s -L %(label)s %(device)s
     label: mylabl3
     filesystem: 'btrfs'
     device: '/dev/xvdh'

# The general format is:
#    fs_setup:
#        - label: <LABEL>
#          filesystem: <FS_TYPE>
#          device: <DEVICE>
#          partition: <PART_VALUE>
#          overwrite: <OVERWRITE>
```

```
#           replace_fs: <FS_TYPE>
#
# Where:
#     <LABEL>: The file system label to be used. If set to None, no label is
#         used.
#
#    <FS_TYPE>: The file system type. It is assumed that the there
#         will be a "mkfs.<FS_TYPE>" that behaves likes "mkfs". On a standard
#         Ubuntu Cloud Image, this means that you have the option of ext{2,3,4},
#         and vfat by default.
#
#    <DEVICE>: The device name. Special names of 'ephemeralX' or 'swap'
#         are allowed and the actual device is acquired from the cloud datasource.
#         When using 'ephemeralX' (i.e. ephemeral0), make sure to leave the
#         label as 'ephemeralX' otherwise there may be issues with the mounting
#         of the ephemeral storage layer.
#
#         If you define the device as 'ephemeralX.Y' then Y will be interpetted
#         as a partition value. However, ephermalX.0 is the _same_ as ephemeralX.
#
#    <PART_VALUE>:
#         Partition definitions are overwriten if you use the '<DEVICE>.Y' notation.
#
#         The valid options are:
#         "auto|any": tell cloud-init not to care whether there is a partition
#             or not. Auto will use the first partition that does not contain a
#             file system already. In the absence of a partition table, it will
#             put it directly on the disk.
#
#             "auto": If a file system that matches the specification in terms of
#             label, type and device, then cloud-init will skip the creation of
#             the file system.
#
#             "any": If a file system that matches the file system type and device,
#             then cloud-init will skip the creation of the file system.
#
#             Devices are selected based on first-detected, starting with partitions
#             and then the raw disk. Consider the following:
#                 NAME     FSTYPE LABEL
#                 xvdb
#                 |-xvdb1  ext4
#                 |-xvdb2
#                 |-xvdb3  btrfs  test
#                 \-xvdb4  ext4   test
#
#             If you ask for 'auto', label of 'test, and file system of 'ext4'
#             then cloud-init will select the 2nd partition, even though there
#             is a partition match at the 4th partition.
#
#             If you ask for 'any' and a label of 'test', then cloud-init will
#             select the 1st partition.
#
#             If you ask for 'auto' and don't define label, then cloud-init will
#             select the 1st partition.
#
#             In general, if you have a specific partition configuration in mind,
#             you should define either the device or the partition number. 'auto'
#             and 'any' are specifically intended for formating ephemeral storage or
```

```
#             for simple schemes.
#
#         "none": Put the file system directly on the device.
#
#         <NUM>: where NUM is the actual partition number.
#
#     <OVERWRITE>: Defines whether or not to overwrite any existing
#         filesystem.
#
#         "true": Indiscriminately destroy any pre-existing file system. Use at
#             your own peril.
#
#         "false": If an existing file system exists, skip the creation.
#
#     <REPLACE_FS>: This is a special directive, used for Windows Azure that
#         instructs cloud-init to replace a file system of <FS_TYPE>. NOTE:
#         unless you define a label, this requires the use of the 'any' partition
#         directive.
#
# Behavior Caveat: The default behavior is to _check_ if the file system exists.
#     If a file system matches the specification, then the operation is a no-op.
```

Grow partitions

```
#cloud-config
#
# growpart entry is a dict, if it is not present at all
# in config, then the default is used ({'mode': 'auto', 'devices': ['/']})
#
#  mode:
#    values:
#     * auto: use any option possible (any available)
#             if none are available, do not warn, but debug.
#     * growpart: use growpart to grow partitions
#             if growpart is not available, this is an error.
#     * off, false
#
# devices:
#   a list of things to resize.
#   items can be filesystem paths or devices (in /dev)
#   examples:
#     devices: [/, /dev/vdb1]
#
# ignore_growroot_disabled:
#   a boolean, default is false.
#   if the file /etc/growroot-disabled exists, then cloud-init will not grow
#   the root partition.  This is to allow a single file to disable both
#   cloud-initramfs-growroot and cloud-init's growroot support.
#
#   true indicates that /etc/growroot-disabled should be ignored
#
growpart:
  mode: auto
  devices: ['/']
  ignore_growroot_disabled: false
```

Boot Stages

In order to be able to provide the functionality that it does, cloud-init must be integrated into the boot in fairly controlled way.

There are 5 stages.

1. **Generator**
2. **Local**
3. **Network**
4. **Config**
5. **Final**

Generator

When booting under systemd, a generator will run that determines if cloud-init.target should be included in the boot goals. By default, this generator will enable cloud-init. It will not enable cloud-init if either:

- A file exists: `/etc/cloud/cloud-init.disabled`
- The kernel command line as found in /proc/cmdline contains `cloud-init=disabled`. When running in a container, the kernel command line is not honored, but cloud-init will read an environment variable named `KERNEL_CMDLINE` in its place.

This mechanism for disabling at runtime currently only exists in systemd.

Local

- **systemd service**: `cloud-init-local.service`
- **runs**: As soon as possible with / mounted read-write.
- **blocks**: as much of boot as possible, *must* block network bringup.
- **modules**: none

The purpose of the local stage is:

- locate "local" data sources.
- apply networking configuration to the system (including "Fallback")

In most cases, this stage does not do much more than that. It finds the datasource and determines the network configuration to be used. That network configuration can come from:

- the datasource
- fallback: Cloud-init's fallback networking consists of rendering the equivalent to "dhcp on eth0", which was historically the most popular mechanism for network configuration of a guest.
- none. network configuration can be disabled entirely with config like the following in /etc/cloud/cloud.cfg: '`network: {config: disabled}`'.

If this is an instance's first boot, then the selected network configuration is rendered. This includes clearing of all previous (stale) configuration including persistent device naming with old mac addresses.

This stage must block network bring-up or any stale configuration might already have been applied. That could have negative effects such as DHCP hooks or broadcast of an old hostname. It would also put the system in an odd state to recover from as it may then have to restart network devices.

Cloud-init then exits and expects for the continued boot of the operating system to bring network configuration up as configured.

Note: In the past, local data sources have been only those that were available without network (such as 'ConfigDrive'). However, as seen in the recent additions to the DigitalOcean datasource, even data sources that require a network can operate at this stage.

Network

- **systemd service**: `cloud-init.service`
- **runs**: After local stage and configured networking is up.
- **blocks**: As much of remaining boot as possible.
- **modules**: `init_modules`

This stage requires all configured networking to be online, as it will fully process any user-data that is found. Here, processing means:

- retrive any `#include` or `#include-once` (recursively) including http
- uncompress any compressed content
- run any part-handler found.

This stage runs the `disk_setup` and `mounts` modules which may partition and format disks and configure mount points (such as in /etc/fstab). Those modules cannot run earlier as they may receive configuration input from sources only available via network. For example, a user may have provided user-data in a network resource that describes how local mounts should be done.

On some clouds such as Azure, this stage will create filesystems to be mounted, including ones that have stale (previous instance) references in /etc/fstab. As such, entries /etc/fstab other than those necessary for cloud-init to run should not be done until after this stage.

A part-handler will run at this stage, as will boothooks including cloud-config `bootcmd`. The user of this functionality has to be aware that the system is in the process of booting when their code runs.

Config

- **systemd service**: `cloud-config.service`
- **runs**: After network stage.
- **blocks**: None.
- **modules**: `config_modules`

This stage runs config modules only. Modules that do not really have an effect on other stages of boot are run here.

Final

- **systemd service**: `cloud-final.service`
- **runs**: As final part of boot (traditional "rc.local")

- **blocks**: None.
- **modules**: `final_modules`

This stage runs as late in boot as possible. Any scripts that a user is accustomed to running after logging into a system should run correctly here. Things that run here include

- package installations
- configuration management plugins (puppet, chef, salt-minion)
- user-scripts (including `runcmd`).

Datasources

What is a datasource?

Datasources are sources of configuration data for cloud-init that typically come from the user (aka userdata) or come from the stack that created the configuration drive (aka metadata). Typical userdata would include files, yaml, and shell scripts while typical metadata would include server name, instance id, display name and other cloud specific details. Since there are multiple ways to provide this data (each cloud solution seems to prefer its own way) internally a datasource abstract class was created to allow for a single way to access the different cloud systems methods to provide this data through the typical usage of subclasses.

The current interface that a datasource object must provide is the following:

```
# returns a mime multipart message that contains
# all the various fully-expanded components that
# were found from processing the raw userdata string
# - when filtering only the mime messages targeting
#   this instance id will be returned (or messages with
#   no instance id)
def get_userdata(self, apply_filter=False)

# returns the raw userdata string (or none)
def get_userdata_raw(self)

# returns a integer (or none) which can be used to identify
# this instance in a group of instances which are typically
# created from a single command, thus allowing programmatic
# filtering on this launch index (or other selective actions)
@property
def launch_index(self)

# the data sources' config_obj is a cloud-config formatted
# object that came to it from ways other than cloud-config
# because cloud-config content would be handled elsewhere
def get_config_obj(self)

#returns a list of public ssh keys
def get_public_ssh_keys(self)

# translates a device 'short' name into the actual physical device
# fully qualified name (or none if said physical device is not attached
# or does not exist)
def device_name_to_device(self, name)

# gets the locale string this instance should be applying
```

```
# which typically used to adjust the instances locale settings files
def get_locale(self)

@property
def availability_zone(self)

# gets the instance id that was assigned to this instance by the
# cloud provider or when said instance id does not exist in the backing
# metadata this will return 'iid-datasource'
def get_instance_id(self)

# gets the fully qualified domain name that this host should  be using
# when configuring network or hostname releated settings, typically
# assigned either by the cloud provider or the user creating the vm
def get_hostname(self, fqdn=False)

def get_package_mirror_info(self)
```

Datasource Documentation

The following is a list of the implemented datasources. Follow for more information.

Alt Cloud

The datasource altcloud will be used to pick up user data on RHEVm and vSphere.

RHEVm

For RHEVm v3.0 the userdata is injected into the VM using floppy injection via the RHEVm dashboard "Custom Properties".

The format of the Custom Properties entry must be:

```
floppyinject=user-data.txt:<base64 encoded data>
```

For example to pass a simple bash script:

```
% cat simple_script.bash
#!/bin/bash
echo "Hello Joe!" >> /tmp/JJV_Joe_out.txt

% base64 < simple_script.bash
IyEvYmluL2Jhc2gKZWNobyAiSGVsbG8gSm9lISIgPj4gL3RtcC9KSlZfSm9lX291dC50eHQK
```

To pass this example script to cloud-init running in a RHEVm v3.0 VM set the "Custom Properties" when creating the RHEMv v3.0 VM to:

```
floppyinject=user-data.
↪txt:IyEvYmluL2Jhc2gKZWNobyAiSGVsbG8gSm9lISIgPj4gL3RtcC9KSlZfSm9lX291dC50eHQK
```

NOTE: The prefix with file name must be: `floppyinject=user-data.txt:`

It is also possible to launch a RHEVm v3.0 VM and pass optional user data to it using the Delta Cloud.

For more information on Delta Cloud see: http://deltacloud.apache.org

vSphere

For VMWare's vSphere the userdata is injected into the VM as an ISO via the cdrom. This can be done using the vSphere dashboard by connecting an ISO image to the CD/DVD drive.

To pass this example script to cloud-init running in a vSphere VM set the CD/DVD drive when creating the vSphere VM to point to an ISO on the data store.

Note: The ISO must contain the user data.

For example, to pass the same `simple_script.bash` to vSphere:

Create the ISO

```
% mkdir my-iso
```

NOTE: The file name on the ISO must be: `user-data.txt`

```
% cp simple_script.bash my-iso/user-data.txt
% genisoimage -o user-data.iso -r my-iso
```

Verify the ISO

```
% sudo mkdir /media/vsphere_iso
% sudo mount -o loop user-data.iso /media/vsphere_iso
% cat /media/vsphere_iso/user-data.txt
% sudo umount /media/vsphere_iso
```

Then, launch the vSphere VM the ISO user-data.iso attached as a CDROM.

It is also possible to launch a vSphere VM and pass optional user data to it using the Delta Cloud.

For more information on Delta Cloud see: http://deltacloud.apache.org

Azure

This datasource finds metadata and user-data from the Azure cloud platform.

Azure Platform

The azure cloud-platform provides initial data to an instance via an attached CD formatted in UDF. That CD contains a 'ovf-env.xml' file that provides some information. Additional information is obtained via interaction with the "endpoint".

To find the endpoint, we now leverage the dhcp client's ability to log its known values on exit. The endpoint server is special DHCP option 245. Depending on your networking stack, this can be done by calling a script in /etc/dhcp/dhclient-exit-hooks or a file in /etc/NetworkManager/dispatcher.d. Both of these call a sub-command 'dhclient_hook' of cloud-init itself. This sub-command will write the client information in json format to /run/cloud-init/dhclient.hook/<interface>.json.

In order for cloud-init to leverage this method to find the endpoint, the cloud.cfg file must contain:

datasource:

Azure: set_hostname: False agent_command: __builtin__

If those files are not available, the fallback is to check the leases file for the endpoint server (again option 245).

You can define the path to the lease file with the 'dhclient_lease_file' configuration. The default value is /var/lib/dhcp/dhclient.eth0.leases.

dhclient_lease_file: /var/lib/dhcp/dhclient.eth0.leases

walinuxagent

In order to operate correctly, cloud-init needs walinuxagent to provide much of the interaction with azure. In addition to "provisioning" code, walinux does the following on the agent is a long running daemon that handles the following things: - generate a x509 certificate and send that to the endpoint

waagent.conf config

in order to use waagent.conf with cloud-init, the following settings are recommended. Other values can be changed or set to the defaults.

```
# disabling provisioning turns off all 'Provisioning.*' function
Provisioning.Enabled=n
# this is currently not handled by cloud-init, so let walinuxagent do it.
ResourceDisk.Format=y
ResourceDisk.MountPoint=/mnt
```

Userdata

Userdata is provided to cloud-init inside the ovf-env.xml file. Cloud-init expects that user-data will be provided as base64 encoded value inside the text child of a element named `UserData` or `CustomData` which is a direct child of the `LinuxProvisioningConfigurationSet` (a sibling to `UserName`) If both `UserData` and `CustomData` are provided behavior is undefined on which will be selected.

In the example below, user-data provided is 'this is my userdata', and the datasource config provided is `{"agent_command": ["start", "walinuxagent"]}`. That agent command will take affect as if it were specified in system config.

Example:

```
<wa:ProvisioningSection>
 <wa:Version>1.0</wa:Version>
 <LinuxProvisioningConfigurationSet
    xmlns="http://schemas.microsoft.com/windowsazure"
    xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
  <ConfigurationSetType>LinuxProvisioningConfiguration</ConfigurationSetType>
  <HostName>myHost</HostName>
  <UserName>myuser</UserName>
  <UserPassword/>
  <CustomData>dGhpcyBpcyBteSB1c2VyZGF0YQ===</CustomData>
  <dscfg>eyJhZ2VudF9jb21tYW5kIjogWyJzdGFydCIsICJ3YWxpbnV4YWdlbnQiXX0=</dscfg>
  <DisableSshPasswordAuthentication>true</DisableSshPasswordAuthentication>
  <SSH>
   <PublicKeys>
    <PublicKey>
     <Fingerprint>6BE7A7C3C8A8F4B123CCA5D0C2F1BE4CA7B63ED7</Fingerprint>
```

```
     <Path>this-value-unused</Path>
    </PublicKey>
   </PublicKeys>
  </SSH>
  </LinuxProvisioningConfigurationSet>
</wa:ProvisioningSection>
```

Configuration

Configuration for the datasource can be read from the system config's or set via the *dscfg* entry in the *LinuxProvisioningConfigurationSet*. Content in dscfg node is expected to be base64 encoded yaml content, and it will be merged into the 'datasource: Azure' entry.

The '`hostname_bounce: command`' entry can be either the literal string 'builtin' or a command to execute. The command will be invoked after the hostname is set, and will have the 'interface' in its environment. If `set_hostname` is not true, then `hostname_bounce` will be ignored.

An example might be: command: ["sh", "-c", "killall dhclient; dhclient $interface"]

```
datasource:
 agent_command
 Azure:
  agent_command: [service, walinuxagent, start]
  set_hostname: True
  hostname_bounce:
   # the name of the interface to bounce
   interface: eth0
   # policy can be 'on', 'off' or 'force'
   policy: on
   # the method 'bounce' command.
   command: "builtin"
   hostname_command: "hostname"
```

hostname

When the user launches an instance, they provide a hostname for that instance. The hostname is provided to the instance in the ovf-env.xml file as `HostName`.

Whatever value the instance provides in its dhcp request will resolve in the domain returned in the 'search' request.

The interesting issue is that a generic image will already have a hostname configured. The ubuntu cloud images have 'ubuntu' as the hostname of the system, and the initial dhcp request on eth0 is not guaranteed to occur after the datasource code has been run. So, on first boot, that initial value will be sent in the dhcp request and *that* value will resolve.

In order to make the `HostName` provided in the ovf-env.xml resolve, a dhcp request must be made with the new value. Walinuxagent (in its current version) handles this by polling the state of hostname and bouncing ('`ifdown eth0; ifup eth0`' the network interface if it sees that a change has been made.

cloud-init handles this by setting the hostname in the DataSource's 'get_data' method via '`hostname $HostName`', and then bouncing the interface. This behavior can be configured or disabled in the datasource config. See 'Configuration' above.

CloudSigma

This datasource finds metadata and user-data from the CloudSigma cloud platform. Data transfer occurs through a virtual serial port of the CloudSigma's VM and the presence of network adapter is **NOT** a requirement, See server context in the public documentation for more information.

Setting a hostname

By default the name of the server will be applied as a hostname on the first boot.

Providing user-data

You can provide user-data to the VM using the dedicated meta field in the server context `cloudinit-user-data`. By default *cloud-config* format is expected there and the `#cloud-config` header could be omitted. However since this is a raw-text field you could provide any of the valid config formats.

You have the option to encode your user-data using Base64. In order to do that you have to add the `cloudinit-user-data` field to the `base64_fields`. The latter is a comma-separated field with all the meta fields whit base64 encoded values.

If your user-data does not need an internet connection you can create a meta field in the server context `cloudinit-dsmode` and set "local" as value. If this field does not exist the default value is "net".

CloudStack

Apache CloudStack expose user-data, meta-data, user password and account sshkey thru the Virtual-Router. For more details on meta-data and user-data, refer the CloudStack Administrator Guide.

URLs to access user-data and meta-data from the Virtual Machine. Here 10.1.1.1 is the Virtual Router IP:

```
http://10.1.1.1/latest/user-data
http://10.1.1.1/latest/meta-data
http://10.1.1.1/latest/meta-data/{metadata type}
```

Configuration

Apache CloudStack datasource can be configured as follows:

```
datasource:
  CloudStack: {}
  None: {}
datasource_list:
  - CloudStack
```

Config Drive

The configuration drive datasource supports the OpenStack configuration drive disk.

See the config drive extension and introduction in the public documentation for more information.

By default, cloud-init does *always* consider this source to be a full-fledged datasource. Instead, the typical behavior is to assume it is really only present to provide networking information. Cloud-init will copy off the network information, apply it to the system, and then continue on. The "full" datasource could then be found in the EC2 metadata service. If this is not the case then the files contained on the located drive must provide equivalents to what the EC2 metadata service would provide (which is typical of the version 2 support listed below)

Version 1

Note: Version 1 is legacy and should be considered deprecated. Version 2 has been supported in OpenStack since 2012.2 (Folsom).

The following criteria are required to as a config drive:

1. Must be formatted with vfat filesystem
2. Must contain *one* of the following files

```
/etc/network/interfaces
/root/.ssh/authorized_keys
/meta.js
```

`/etc/network/interfaces`

> This file is laid down by nova in order to pass static networking information to the guest. Cloud-init will copy it off of the config-drive and into /etc/network/interfaces (or convert it to RH format) as soon as it can, and then attempt to bring up all network interfaces.

`/root/.ssh/authorized_keys`

> This file is laid down by nova, and contains the ssk keys that were provided to nova on instance creation (nova-boot –key ....)

`/meta.js`

> meta.js is populated on the config-drive in response to the user passing "meta flags" (nova boot –meta key=value ...). It is expected to be json formatted.

Version 2

The following criteria are required to as a config drive:

1. Must be formatted with vfat or iso9660 filesystem or have a *filesystem* label of **config-2**
2. The files that will typically be present in the config drive are:

```
openstack/
  - 2012-08-10/ or latest/
    - meta_data.json
    - user_data (not mandatory)
  - content/
    - 0000 (referenced content files)
    - 0001
    - ....
ec2
  - latest/
    - meta-data.json (not mandatory)
```

Keys and values

Cloud-init's behavior can be modified by keys found in the meta.js (version 1 only) file in the following ways.

```
dsmode:
  values: local, net, pass
  default: pass
```

This is what indicates if configdrive is a final data source or not. By default it is 'pass', meaning this datasource should not be read. Set it to 'local' or 'net' to stop cloud-init from continuing on to search for other data sources after network config.

The difference between 'local' and 'net' is that local will not require networking to be up before user-data actions (or boothooks) are run.

```
instance-id:
  default: iid-dsconfigdrive
```

This is utilized as the metadata's instance-id. It should generally be unique, as it is what is used to determine "is this a new instance".

```
public-keys:
  default: None
```

If present, these keys will be used as the public keys for the instance. This value overrides the content in authorized_keys.

Note: it is likely preferable to provide keys via user-data

```
user-data:
  default: None
```

This provides cloud-init user-data. See *examples* for what all can be present here.

Digital Ocean

The DigitalOcean datasource consumes the content served from DigitalOcean's metadata service. This metadata service serves information about the running droplet via HTTP over the link local address 169.254.169.254. The metadata API endpoints are fully described at https://developers.digitalocean.com/metadata/.

Configuration

DigitalOcean's datasource can be configured as follows:

datasource:

DigitalOcean: retries: 3 timeout: 2

- *retries*: Determines the number of times to attempt to connect to the metadata service
- *timeout*: Determines the timeout in seconds to wait for a response from the metadata service

Amazon EC2

The EC2 datasource is the oldest and most widely used datasource that cloud-init supports. This datasource interacts with a *magic* ip that is provided to the instance by the cloud provider. Typically this ip is `169.254.169.254` of

which at this ip a http server is provided to the instance so that the instance can make calls to get instance userdata and instance metadata.

Metadata is accessible via the following URL:

```
GET http://169.254.169.254/2009-04-04/meta-data/
ami-id
ami-launch-index
ami-manifest-path
block-device-mapping/
hostname
instance-id
instance-type
local-hostname
local-ipv4
placement/
public-hostname
public-ipv4
public-keys/
reservation-id
security-groups
```

Userdata is accessible via the following URL:

```
GET http://169.254.169.254/2009-04-04/user-data
1234,fred,reboot,true | 4512,jimbo, | 173,,,
```

Note that there are multiple versions of this data provided, cloud-init by default uses **2009-04-04** but newer versions can be supported with relative ease (newer versions have more data exposed, while maintaining backward compatibility with the previous versions).

To see which versions are supported from your cloud provider use the following URL:

```
GET http://169.254.169.254/
1.0
2007-01-19
2007-03-01
2007-08-29
2007-10-10
2007-12-15
2008-02-01
2008-09-01
2009-04-04
...
latest
```

MAAS

TODO

For now see: http://maas.ubuntu.com/

NoCloud

The data source `NoCloud` allows the user to provide user-data and meta-data to the instance without running a network service (or even without having a network at all).

You can provide meta-data and user-data to a local vm boot via files on a vfat or iso9660 filesystem. The filesystem volume label must be `cidata`.

Alternatively, you can provide meta-data via kernel command line or SMBIOS "serial number" option. The data must be passed in the form of a string:

```
ds=nocloud[;key=val;key=val]
```

or

```
ds=nocloud-net[;key=val;key=val]
```

The permitted keys are:

- `h` or `local-hostname`
- `i` or `instance-id`
- `s` or `seedfrom`

With `ds=nocloud`, the `seedfrom` value must start with `/` or `file://`. With `ds=nocloud-net`, the `seedfrom` value must start with `http://`, `https://` or `ftp://`

e.g. you can pass this option to QEMU:

```
-smbios type=1,serial=ds=nocloud-net;s=http://10.10.0.1:8000/
```

to cause NoCloud to fetch the full meta-data from http://10.10.0.1:8000/meta-data after the network initialization is complete.

These user-data and meta-data files are expected to be in the following format.

```
/user-data
/meta-data
```

Basically, user-data is simply user-data and meta-data is a yaml formatted file representing what you'd find in the EC2 metadata service.

Given a disk ubuntu 12.04 cloud image in 'disk.img', you can create a sufficient disk by following the example below.

```
## create user-data and meta-data files that will be used
## to modify image on first boot
$ { echo instance-id: iid-local01; echo local-hostname: cloudimg; } > meta-data

$ printf "#cloud-config\npassword: passw0rd\nchpasswd: { expire: False }\nssh_pwauth:␣
↪True\n" > user-data

## create a disk to attach with some user-data and meta-data
$ genisoimage  -output seed.iso -volid cidata -joliet -rock user-data meta-data

## alternatively, create a vfat filesystem with same files
## $ truncate --size 2M seed.img
## $ mkfs.vfat -n cidata seed.img
## $ mcopy -oi seed.img user-data meta-data ::

## create a new qcow image to boot, backed by your original image
$ qemu-img create -f qcow2 -b disk.img boot-disk.img

## boot the image and login as 'ubuntu' with password 'passw0rd'
## note, passw0rd was set as password through the user-data above,
## there is no password set on these images.
```

```
$ kvm -m 256 \
   -net nic -net user,hostfwd=tcp::2222-:22 \
   -drive file=boot-disk.img,if=virtio \
   -drive file=seed.iso,if=virtio
```

Note: that the instance-id provided (`iid-local01` above) is what is used to determine if this is "first boot". So if you are making updates to user-data you will also have to change that, or start the disk fresh.

Also, you can inject an `/etc/network/interfaces` file by providing the content for that file in the `network-interfaces` field of metadata.

Example metadata:

```
instance-id: iid-abcdefg
network-interfaces: |
  iface eth0 inet static
  address 192.168.1.10
  network 192.168.1.0
  netmask 255.255.255.0
  broadcast 192.168.1.255
  gateway 192.168.1.254
hostname: myhost
```

Network configuration can also be provided to cloud-init in either *Networking Config Version 1* or *Networking Config Version 2* by providing that yaml formatted data in a file named `network-config`. If found, this file will override a `network-interfaces` file.

See an example below. Note specifically that this file does not have a top level `network` key as it it is already assumed to be network configuration based on the filename.

```
version: 1
config:
   - type: physical
     name: interface0
     mac_address: "52:54:00:12:34:00"
     subnets:
        - type: static
          address: 192.168.1.10
          netmask: 255.255.255.0
          gateway: 192.168.1.254
```

```
version: 2
ethernets:
  interface0:
     match:
         mac_address: "52:54:00:12:34:00"
     set-name: interface0
     addresses:
     - 192.168.1.10/255.255.255.0
     gateway4: 192.168.1.254
```

OpenNebula

The OpenNebula (ON) datasource supports the contextualization disk.

> See contextualization overview, contextualizing VMs and network configuration in the public documentation for more information.

OpenNebula's virtual machines are contextualized (parametrized) by CD-ROM image, which contains a shell script *context.sh* with custom variables defined on virtual machine start. There are no fixed contextualization variables, but the datasource accepts many used and recommended across the documentation.

Datasource configuration

Datasource accepts following configuration options.

```
dsmode:
  values: local, net, disabled
  default: net
```

Tells if this datasource will be processed in 'local' (pre-networking) or 'net' (post-networking) stage or even completely 'disabled'.

```
parseuser:
  default: nobody
```

Unprivileged system user used for contextualization script processing.

Contextualization disk

The following criteria are required:

1. Must be formatted with iso9660 filesystem or have a *filesystem* label of **CONTEXT** or **CDROM**
2. Must contain file *context.sh* with contextualization variables. File is generated by OpenNebula, it has a KEY='VALUE' format and can be easily read by bash

Contextualization variables

There are no fixed contextualization variables in OpenNebula, no standard. Following variables were found on various places and revisions of the OpenNebula documentation. Where multiple similar variables are specified, only first found is taken.

```
DSMODE
```

Datasource mode configuration override. Values: local, net, disabled.

```
DNS
ETH<x>_IP
ETH<x>_NETWORK
ETH<x>_MASK
ETH<x>_GATEWAY
ETH<x>_DOMAIN
ETH<x>_DNS
```

Static network configuration.

```
HOSTNAME
```

Instance hostname.

```
PUBLIC_IP
IP_PUBLIC
ETH0_IP
```

If no hostname has been specified, cloud-init will try to create hostname from instance's IP address in 'local' dsmode. In 'net' dsmode, cloud-init tries to resolve one of its IP addresses to get hostname.

```
SSH_KEY
SSH_PUBLIC_KEY
```

One or multiple SSH keys (separated by newlines) can be specified.

```
USER_DATA
USERDATA
```

cloud-init user data.

Example configuration

This example cloud-init configuration (*cloud.cfg*) enables OpenNebula datasource only in 'net' mode.

```
disable_ec2_metadata: True
datasource_list: ['OpenNebula']
datasource:
  OpenNebula:
    dsmode: net
    parseuser: nobody
```

Example VM's context section

```
CONTEXT=[
  PUBLIC_IP="$NIC[IP]",
  SSH_KEY="$USER[SSH_KEY]
$USER[SSH_KEY1]
$USER[SSH_KEY2] ",
  USER_DATA="#cloud-config
# see https://help.ubuntu.com/community/CloudInit

packages: []

mounts:
- [vdc,none,swap,sw,0,0]
runcmd:
- echo 'Instance has been configured by cloud-init.' | wall
" ]
```

OpenStack

This datasource supports reading data from the OpenStack Metadata Service.

Configuration

The following configuration can be set for the datasource in system configuration (in */etc/cloud/cloud.cfg* or */etc/cloud/cloud.cfg.d/*).

The settings that may be configured are:

- **metadata_urls**: This list of urls will be searched for an OpenStack metadata service. The first entry that successfully returns a 200 response for <url>/openstack will be selected. (default: ['http://169.254.169.254']).
- **max_wait**: the maximum amount of clock time in seconds that should be spent searching metadata_urls. A value less than zero will result in only one request being made, to the first in the list. (default: -1)
- **timeout**: the timeout value provided to urlopen for each individual http request. This is used both when selecting a metadata_url and when crawling the metadata service. (default: 10)
- **retries**: The number of retries that should be done for an http request. This value is used only after metadata_url is selected. (default: 5)

An example configuration with the default values is provided as example below:

```
#cloud-config
datasource:
 OpenStack:
  metadata_urls: ["http://169.254.169.254"]
  max_wait: -1
  timeout: 10
  retries: 5
```

Vendor Data

The OpenStack metadata server can be configured to serve up vendor data which is available to all instances for consumption. OpenStack vendor data is, generally, a JSON object.

cloud-init will look for configuration in the `cloud-init` attribute of the vendor data JSON object. cloud-init processes this configuration using the same handlers as user data, so any formats that work for user data should work for vendor data.

For example, configuring the following as vendor data in OpenStack would upgrade packages and install `htop` on all instances:

```
{"cloud-init": "#cloud-config\npackage_upgrade: True\npackages:\n - htop"}
```

For more general information about how cloud-init handles vendor data, including how it can be disabled by users on instances, see *Vendor Data*.

OVF

The OVF Datasource provides a datasource for reading data from on an Open Virtualization Format ISO transport.

For further information see a full working example in cloud-init's source code tree in doc/sources/ovf

SmartOS Datasource

This datasource finds metadata and user-data from the SmartOS virtualization platform (i.e. Joyent).

Please see http://smartos.org/ for information about SmartOS.

SmartOS Platform

The SmartOS virtualization platform uses meta-data to the instance via the second serial console. On Linux, this is /dev/ttyS1. The data is a provided via a simple protocol: something queries for the data, the console responds responds with the status and if "SUCCESS" returns until a single ".n".

New versions of the SmartOS tooling will include support for base64 encoded data.

Meta-data channels

Cloud-init supports three modes of delivering user/meta-data via the flexible channels of SmartOS.

- user-data is written to /var/db/user-data
  - per the spec, user-data is for consumption by the end-user, not provisioning tools
  - cloud-init entirely ignores this channel other than writting it to disk
  - removal of the meta-data key means that /var/db/user-data gets removed
  - a backup of previous meta-data is maintained as /var/db/user-data.<timestamp>. <timestamp> is the epoch time when cloud-init ran
- user-script is written to /var/lib/cloud/scripts/per-boot/99_user_data
  - this is executed each boot
  - a link is created to /var/db/user-script
  - previous versions of the user-script is written to /var/lib/cloud/scripts/per-boot.backup/99_user_script.<timestamp>. - <timestamp> is the epoch time when cloud-init ran.
  - when the 'user-script' meta-data key goes missing, the user-script is removed from the file system, although a backup is maintained.
  - if the script is not shebanged (i.e. starts with #!<executable>), then or is not an executable, cloud-init will add a shebang of "#!/bin/bash"
- cloud-init:user-data is treated like on other Clouds.
  - this channel is used for delivering _all_ cloud-init instructions
  - scripts delivered over this channel must be well formed (i.e. must have a shebang)

Cloud-init supports reading the traditional meta-data fields supported by the SmartOS tools. These are:

- root_authorized_keys
- hostname
- enable_motd_sys_info
- iptables_disable

Note: At this time iptables_disable and enable_motd_sys_info are read but are not actioned.

Disabling user-script

Cloud-init uses the per-boot script functionality to handle the execution of the user-script. If you want to prevent this use a cloud-config of:

```
#cloud-config
cloud_final_modules:
 - scripts-per-once
 - scripts-per-instance
 - scripts-user
 - ssh-authkey-fingerprints
 - keys-to-console
 - phone-home
 - final-message
 - power-state-change
```

Alternatively you can use the json patch method

```
#cloud-config-jsonp
[
     { "op": "replace",
       "path": "/cloud_final_modules",
       "value": ["scripts-per-once",
                 "scripts-per-instance",
                 "scripts-user",
                 "ssh-authkey-fingerprints",
                 "keys-to-console",
                 "phone-home",
                 "final-message",
                 "power-state-change"]
     }
]
```

The default cloud-config includes "script-per-boot". Cloud-init will still ingest and write the user-data but will not execute it, when you disable the per-boot script handling.

Note: Unless you have an explicit use-case, it is recommended that you not disable the per-boot script execution, especially if you are using any of the life-cycle management features of SmartOS.

The cloud-config needs to be delivered over the cloud-init:user-data channel in order for cloud-init to ingest it.

base64

The following are exempt from base64 encoding, owing to the fact that they are provided by SmartOS:

- root_authorized_keys
- enable_motd_sys_info
- iptables_disable
- user-data
- user-script

This list can be changed through system config of variable 'no_base64_decode'.

This means that user-script and user-data as well as other values can be base64 encoded. Since Cloud-init can only guess as to whether or not something is truly base64 encoded, the following meta-data keys are hints as to whether or not to base64 decode something:

- base64_all: Except for excluded keys, attempt to base64 decode the values. If the value fails to decode properly, it will be returned in its text
- base64_keys: A comma deliminated list of which keys are base64 encoded.

- b64-<key>: for any key, if there exists an entry in the metadata for 'b64-<key>' Then 'b64-<key>' is expected to be a plaintext boolean indicating whether or not its value is encoded.
- no_base64_decode: This is a configuration setting (i.e. /etc/cloud/cloud.cfg.d) that sets which values should not be base64 decoded.

disk_aliases and ephemeral disk

By default, SmartOS only supports a single ephemeral disk. That disk is completely empty (un-partitioned with no filesystem).

The SmartOS datasource has built-in cloud-config which instructs the 'disk_setup' module to partition and format the ephemeral disk.

You can control the disk_setup then in 2 ways:

1. through the datasource config, you can change the 'alias' of ephermeral0 to reference another device. The default is:

   'disk_aliases': {'ephemeral0': '/dev/vdb'},

   Which means anywhere disk_setup sees a device named 'ephemeral0' then /dev/vdb will be substituted.

2. you can provide disk_setup or fs_setup data in user-data to overwrite the datasource's built-in values.

See doc/examples/cloud-config-disk-setup.txt for information on disk_setup.

Fallback/None

This is the fallback datasource when no other datasource can be selected. It is the equivalent of a empty datasource in that it provides a empty string as userdata and a empty dictionary as metadata. It is useful for testing as well as for when you do not have a need to have an actual datasource to meet your instance requirements (ie you just want to run modules that are not concerned with any external data). It is typically put at the end of the datasource search list so that if all other datasources are not matched, then this one will be so that the user is not left with an inaccessible instance.

Note: the instance id that this datasource provides is `iid-datasource-none`.

Google Compute Engine

The GCE datasource gets its data from the internal compute metadata server. Metadata can be queried at the URL 'http://metadata.google.internal/computeMetadata/v1/' from within an instance. For more information see the GCE metadata docs.

Currently the default project and instance level metadatakeys keys `project/attributes/sshKeys` and `instance/attributes/ssh-keys` are merged to provide `public-keys`.

`user-data` and `user-data-encoding` can be provided to cloud-init by setting those custom metadata keys for an *instance*.

Logging

Cloud-init supports both local and remote logging configurable through python's built-in logging configuration and through the cloud-init rsyslog module.

Command Output

Cloud-init can redirect its stdout and stderr based on config given under the `output` config key. The output of any commands run by cloud-init and any user or vendor scripts provided will also be included here. The `output` key accepts a dictionary for configuration. Output files may be specified individually for each stage (`init`, `config`, and `final`), or a single key `all` may be used to specify output for all stages.

The output for each stage may be specified as a dictionary of `output` and `error` keys, for stdout and stderr respectively, as a tuple with stdout first and stderr second, or as a single string to use for both. The strings passed to all of these keys are handled by the system shell, so any form of redirection that can be used in bash is valid, including piping cloud-init's output to `tee`, or `logger`. If only a filename is provided, cloud-init will append its output to the file as though >> was specified.

By default, cloud-init loads its output configuration from `/etc/cloud/cloud.cfg.d/05_logging.cfg`. The default config directs both stdout and stderr from all cloud-init stages to `/var/log/cloud-init-output.log`. The default config is given as

```
output: { all: "| tee -a /var/log/cloud-init-output.log" }
```

For a more complex example, the following configuration would output the init stage to `/var/log/cloud-init.out` and `/var/log/cloud-init.err`, for stdout and stderr respectively, replacing anything that was previously there. For the config stage, it would pipe both stdout and stderr through `tee -a /var/log/cloud-config.log`. For the final stage it would append the output of stdout and stderr to `/var/log/cloud-final.out` and `/var/log/cloud-final.err` respectively.

```
output:
    init:
        output: "> /var/log/cloud-init.out"
        error: "> /var/log/cloud-init.err"
    config: "tee -a /var/log/cloud-config.log"
    final:
        - ">> /var/log/cloud-final.out"
        - "/var/log/cloud-final.err"
```

Python Logging

Cloud-init uses the python logging module, and can accept config for this module using the standard python fileConfig format. Cloud-init looks for config for the logging module under the `logcfg` key.

Note: the logging configuration is not yaml, it is python `fileConfig` format, and is passed through directly to the python logging module. please use the correct syntax for a multi-line string in yaml.

By default, cloud-init uses the logging configuration provided in `/etc/cloud/cloud.cfg.d/05_logging.cfg`. The default python logging configuration writes all cloud-init events with a priority of `WARNING` or higher to console, and writes all events with a level of `DEBUG` or higher to `/var/log/cloud-init.log` and via syslog.

Python's fileConfig format consists of sections with headings in the format `[title]` and key value pairs in each section. Configuration for python logging must contain the sections `[loggers]`, `[handlers]`, and `[formatters]`, which name the entities of their respective types that will be defined. The section name for each defined logger, handler and formatter will start with its type, followed by an underscore (_) and the name of the entity. For example, if a logger was specified with the name `log01`, config for the logger would be in the section `[logger_log01]`.

Logger config entries contain basic logging set up. They may specify a list of handlers to send logging events to as well as the lowest priority level of events to handle. A logger named `root` must be specified and its configuration (under `[logger_root]`) must contain a level and a list of handlers. A level entry can be any of the following:

DEBUG, INFO, WARNING, ERROR, CRITICAL, or NOTSET. For the root logger the NOTSET option will allow all logging events to be recorded.

Each configured handler must specify a class under the python's logging package namespace. A handler may specify a message formatter to use, a priority level, and arguments for the handler class. Common handlers are StreamHandler, which handles stream redirects (i.e. logging to stderr), and FileHandler which outputs to a log file. The logging module also supports logging over net sockets, over http, via smtp, and additional complex configurations. For full details about the handlers available for python logging, please see the documentation for python logging handlers.

Log messages are formatted using the logging.Formatter class, which is configured using formatter config entities. A default format of %(message)s is given if no formatter configs are specified. Formatter config entities accept a format string which supports variable replacements. These may also accept a datefmt string which may be used to configure the timestamp used in the log messages. The format variables %(asctime)s, %(levelname)s and %(message)s are commonly used and represent the timestamp, the priority level of the event and the event message. For additional information on logging formatters see python logging formatters.

Note: by default the format string used in the logging formatter are in python's old style %s form. the str.format() and string.Template styles can also be used by using { or $ in place of % by setting the style parameter in formatter config.

A simple, but functional python logging configuration for cloud-init is below. It will log all messages of priority DEBUG or higher both stderr and /tmp/my.log using a StreamHandler and a FileHandler, using the default format string %(message)s:

```
logcfg: |
 [loggers]
 keys=root,cloudinit
 [handlers]
 keys=ch,cf
 [formatters]
 keys=
 [logger_root]
 level=DEBUG
 handlers=
 [logger_cloudinit]
 level=DEBUG
 qualname=cloudinit
 handlers=ch,cf
 [handler_ch]
 class=StreamHandler
 level=DEBUG
 args=(sys.stderr,)
 [handler_cf]
 class=FileHandler
 level=DEBUG
 args=('/tmp/my.log',)
```

For additional information about configuring python's logging module, please see the documentation for python logging config.

Rsyslog Module

Cloud-init's cc_rsyslog module allows for fully customizable rsyslog configuration under the rsyslog config key. The simplest way to use the rsyslog module is by specifying remote servers under the remotes key in rsyslog

config. The `remotes` key takes a dictionary where each key represents the name of an rsyslog server and each value is the configuration for that server. The format for server config is:

- optional filter for log messages (defaults to `*.*`)
- optional leading `@` or `@@`, indicating udp and tcp respectively (defaults to `@`, for udp)
- ipv4 or ipv6 hostname or address. ipv6 addresses must be in `[::1]` format, (e.g. `@[fd00::1]:514`)
- optional port number (defaults to `514`)

For example, to send logging to an rsyslog server named `log_serv` with address `10.0.4.1`, using port number `514`, over udp, with all log messages enabled one could use either of the following.

With all options specified:

```
rsyslog:
    remotes:
        log_serv: "*.* @10.0.4.1:514"
```

With defaults used:

```
rsyslog:
    remotes:
        log_serv: "10.0.4.1"
```

For more information on rsyslog configuration, see *Rsyslog*.

Modules

Apt Configure

Summary: configure apt

This module handles both configuration of apt options and adding source lists. There are configuration options such as `apt_get_wrapper` and `apt_get_command` that control how cloud-init invokes apt-get. These configuration options are handled on a per-distro basis, so consult documentation for cloud-init's distro support for instructions on using these config options.

Note: To ensure that apt configuration is valid yaml, any strings containing special characters, especially `:` should be quoted.

Note: For more information about apt configuration, see the `Additional apt configuration` example.

Preserve sources.list:

By default, cloud-init will generate a new sources list in `/etc/apt/sources.list.d` based on any changes specified in cloud config. To disable this behavior and preserve the sources list from the pristine image, set `preserve_sources_list` to `true`.

Note: The `preserve_sources_list` option overrides all other config keys that would alter `sources.list` or `sources.list.d`, **except** for additional sources to be added to `sources.list.d`.

Disable source suites:

Entries in the sources list can be disabled using `disable_suites`, which takes a list of suites to be disabled. If the string `$RELEASE` is present in a suite in the `disable_suites` list, it will be replaced with the release name. If a suite specified in `disable_suites` is not present in `sources.list` it will be ignored. For convenience, several aliases are provided for `disable_suites`:

- `updates` => `$RELEASE-updates`
- `backports` => `$RELEASE-backports`
- `security` => `$RELEASE-security`
- `proposed` => `$RELEASE-proposed`
- `release` => `$RELEASE`

Note: When a suite is disabled using `disable_suites`, its entry in `sources.list` is not deleted; it is just commented out.

Configure primary and security mirrors:

The primary and security archive mirrors can be specified using the `primary` and `security` keys, respectively. Both the `primary` and `security` keys take a list of configs, allowing mirrors to be specified on a per-architecture basis. Each config is a dictionary which must have an entry for `arches`, specifying which architectures that config entry is for. The keyword `default` applies to any architecture not explicitly listed. The mirror url can be specified with the `uri` key, or a list of mirrors to check can be provided in order, with the first mirror that can be resolved being selected. This allows the same configuration to be used in different environment, with different hosts used for a local apt mirror. If no mirror is provided by `uri` or `search`, `search_dns` may be used to search for dns names in the format `<distro>-mirror` in each of the following:

- fqdn of this host per cloud metadata
- localdomain
- domains listed in `/etc/resolv.conf`

If there is a dns entry for `<distro>-mirror`, then it is assumed that there is a distro mirror at `http://<distro>-mirror.<domain>/<distro>`. If the `primary` key is defined, but not the `security` key, then then configuration for `primary` is also used for `security`. If `search_dns` is used for the `security` key, the search pattern will be. `<distro>-security-mirror`.

If no mirrors are specified, or all lookups fail, then default mirrors defined in the datasource are used. If none are present in the datasource either the following defaults are used:

- primary: `http://archive.ubuntu.com/ubuntu`
- security: `http://security.ubuntu.com/ubuntu`

Specify sources.list template:

A custom template for rendering `sources.list` can be specefied with `sources_list`. If no `sources_list` template is given, cloud-init will use sane default. Within this template, the following strings will be replaced with the appropriate values:

- `$MIRROR`
- `$RELEASE`
- `$PRIMARY`
- `$SECURITY`

Pass configuration to apt:

Apt configuration can be specified using `conf`. Configuration is specified as a string. For multiline apt configuration, make sure to follow yaml syntax.

Configure apt proxy:

Proxy configuration for apt can be specified using `conf`, but proxy config keys also exist for convenience. The proxy config keys, `http_proxy`, `ftp_proxy`, and `https_proxy` may be used to specify a proxy for http, ftp and https protocols respectively. The `proxy` key also exists as an alias for `http_proxy`. Proxy url is specified in the format `<protocol>://[[user][:pass]@]host[:port]/`.

Add apt repos by regex:

All source entries in `apt-sources` that match regex in `add_apt_repo_match` will be added to the system using `add-apt-repository`. If `add_apt_repo_match` is not specified, it defaults to `^[\w-]+:\w`

Add source list entries:

Source list entries can be specified as a dictionary under the `sources` config key, with key in the dict representing a different source file. The key The key of each source entry will be used as an id that can be referenced in other config entries, as well as the filename for the source's configuration under `/etc/apt/sources.list.d`. If the name does not end with `.list`, it will be appended. If there is no configuration for a key in `sources`, no file will be written, but the key may still be referred to as an id in other `sources` entries.

Each entry under `sources` is a dictionary which may contain any of the following optional keys:

- `source`: a sources.list entry (some variable replacements apply)
- `keyid`: a key to import via shortid or fingerprint
- `key`: a raw PGP key
- `keyserver`: alternate keyserver to pull `keyid` key from

The `source` key supports variable replacements for the following strings:

- `$MIRROR`
- `$PRIMARY`
- `$SECURITY`
- `$RELEASE`

Internal name: `cc_apt_configure`

Module frequency: per instance

Supported distros: ubuntu, debian

Config keys:

```
apt:
    preserve_sources_list: <true/false>
    disable_suites:
        - $RELEASE-updates
        - backports
        - $RELEASE
        - mysuite
    primary:
        - arches:
            - amd64
            - i386
            - default
```

```
        uri: "http://us.archive.ubuntu.com/ubuntu"
        search:
          - "http://cool.but-sometimes-unreachable.com/ubuntu"
          - "http://us.archive.ubuntu.com/ubuntu"
        search_dns: <true/false>
      - arches:
          - s390x
          - arm64
        uri: "http://archive-to-use-for-arm64.example.com/ubuntu"
  security:
      - arches:
          - default
        search_dns: true
  sources_list: |
      deb $MIRROR $RELEASE main restricted
      deb-src $MIRROR $RELEASE main restricted
      deb $PRIMARY $RELEASE universe restricted
      deb $SECURITY $RELEASE-security multiverse
  debconf_selections:
      set1: the-package the-package/some-flag boolean true
  conf: |
      APT {
          Get {
              Assume-Yes "true";
              Fix-Broken "true";
          }
      }
  proxy: "http://[[user][:pass]@]host[:port]/"
  http_proxy: "http://[[user][:pass]@]host[:port]/"
  ftp_proxy: "ftp://[[user][:pass]@]host[:port]/"
  https_proxy: "https://[[user][:pass]@]host[:port]/"
  sources:
      source1:
          keyid: "keyid"
          keyserver: "keyserverurl"
          source: "deb http://<url>/ xenial main"
      source2:
          source: "ppa:<ppa-name>"
      source3:
          source: "deb $MIRROR $RELEASE multiverse"
          key: |
              ------BEGIN PGP PUBLIC KEY BLOCK-------
              <key data>
              ------END PGP PUBLIC KEY BLOCK-------
```

Apt Pipelining

Summary: configure apt pipelining

This module configures apt's `Acquite::http::Pipeline-Depth` option, whcih controls how apt handles HTTP pipelining. It may be useful for pipelining to be disabled, because some web servers, such as S3 do not pipeline properly (LP: #948461). The `apt_pipelining` config key may be set to `false` to disable pipelining altogether. This is the default behavior. If it is set to `none`, `unchanged`, or `os`, no change will be made to apt configuration and the default setting for the distro will be used. The pipeline depth can also be manually specified by setting `apt_pipelining` to a number. However, this is not recommended.

Internal name: `cc_apt_pipelining`

Module frequency: per instance

Supported distros: ubuntu, debian

Config keys:: apt_pipelining: <false/none/unchanged/os/number>

Bootcmd

Summary: Run arbitrary commands early in the boot process

This module runs arbitrary commands very early in the boot process, only slightly after a boothook would run. This is very similar to a boothook, but more user friendly. The environment variable `INSTANCE_ID` will be set to the current instance id for all run commands. Commands can be specified either as lists or strings. For invocation details, see `runcmd`.

Note: bootcmd should only be used for things that could not be done later in the boot process.

Internal name: `cc_bootcmd`

Module frequency: always

Supported distros: all

Config schema: **bootcmd:** (array of (array of string)/(string))

Examples:

```
bootcmd:
    - echo 192.168.1.130 us.archive.ubuntu.com > /etc/hosts
    - [ cloud-init-per, once, mymkfs, mkfs, /dev/vdb ]
```

Byobu

Summary: enable/disable byobu system wide and for default user

This module controls whether byobu is enabled or disabled system wide and for the default system user. If byobu is to be enabled, this module will ensure it is installed. Likewise, if it is to be disabled, it will be removed if installed.

Valid configuration options for this module are:

- `enable-system`: enable byobu system wide
- `enable-user`: enable byobu for the default user
- `disable-system`: disable byobu system wide
- `disable-user`: disable byobu for the default user
- `enable`: enable byobu both system wide and for default user
- `disable`: disable byobu for all users
- `user`: alias for `enable-user`
- `system`: alias for `enable-system`

Internal name: `cc_byobu`

Module frequency: per instance

Supported distros: ubuntu, debian

Config keys:

```
byobu_by_default: <user/system>
```

CA Certs

Summary: add ca certificates

This module adds CA certificates to `/etc/ca-certificates.conf` and updates the ssl cert cache using `update-ca-certificates`. The default certificates can be removed from the system with the configuration option `remove-defaults`.

Note: certificates must be specified using valid yaml. in order to specify a multiline certificate, the yaml multiline list syntax must be used

---

Internal name: `cc_ca_certs`

Module frequency: per instance

Supported distros: ubuntu, debian

Config keys:

```
ca-certs:
    remove-defaults: <true/false>
    trusted:
        - <single line cert>
        - |
          -----BEGIN CERTIFICATE-----
          YOUR-ORGS-TRUSTED-CA-CERT-HERE
          -----END CERTIFICATE-----
```

Chef

Summary: module that configures, starts and installs chef.

This module enables chef to be installed (from packages or from gems, or from omnibus). Before this occurs chef configurations are written to disk (validation.pem, client.pem, firstboot.json, client.rb), and needed chef folders/directories are created (/etc/chef and /var/log/chef and so-on). Then once installing proceeds correctly if configured chef will be started (in daemon mode or in non-daemon mode) and then once that has finished (if ran in non-daemon mode this will be when chef finishes converging, if ran in daemon mode then no further actions are possible since chef will have forked into its own process) then a post run function can run that can do finishing activities (such as removing the validation pem file).

Internal name: `cc_chef`

Module frequency: per always

Supported distros: all

Config keys:

```
chef:
   directories: (defaulting to /etc/chef, /var/log/chef, /var/lib/chef,
                 /var/cache/chef, /var/backups/chef, /var/run/chef)
   validation_cert: (optional string to be written to file validation_key)
```

```
                    special value 'system' means set use existing file
   validation_key: (optional the path for validation_cert. default
                    /etc/chef/validation.pem)
   firstboot_path: (path to write run_list and initial_attributes keys that
                    should also be present in this configuration, defaults
                    to /etc/chef/firstboot.json)
   exec: boolean to run or not run chef (defaults to false, unless
                                         a gem installed is requested
                                         where this will then default
                                         to true)

chef.rb template keys (if falsey, then will be skipped and not
                       written to /etc/chef/client.rb)

chef:
  client_key:
  environment:
  file_backup_path:
  file_cache_path:
  json_attribs:
  log_level:
  log_location:
  node_name:
  omnibus_url:
  omnibus_url_retries:
  omnibus_version:
  pid_file:
  server_url:
  show_time:
  ssl_verify_mode:
  validation_cert:
  validation_key:
  validation_name:
```

Debug

Summary: helper to debug cloud-init *internal* datastructures.

This module will enable for outputting various internal information that cloud-init sources provide to either a file or to the output console/log location that this cloud-init has been configured with when running.

Note: Log configurations are not output.

Internal name: `cc_debug`

Module frequency: per instance

Supported distros: all

Config keys:

```
debug:
   verbose: true/false (defaulting to true)
   output: (location to write output, defaulting to console + log)
```

Disable EC2 Metadata

Summary: disable aws ec2 metadata

This module can disable the ec2 datasource by rejecting the route to `169.254.169.254`, the usual route to the datasource. This module is disabled by default.

Internal name: `cc_disable_ec2_metadata`

Module frequency: per always

Supported distros: all

Config keys:

```
disable_ec2_metadata: <true/false>
```

Disk Setup

Summary: configure partitions and filesystems

This module is able to configure simple partition tables and filesystems.

Note: for more detail about configuration options for disk setup, see the disk setup example

---

For convenience, aliases can be specified for disks using the `device_aliases` config key, which takes a dictionary of alias: path mappings. There are automatic aliases for `swap` and `ephemeral<X>`, where `swap` will always refer to the active swap partition and `ephemeral<X>` will refer to the block device of the ephemeral image.

Disk partitioning is done using the `disk_setup` directive. This config directive accepts a dictionary where each key is either a path to a block device or an alias specified in `device_aliases`, and each value is the configuration options for the device. The `table_type` option specifies the partition table type, either `mbr` or `gpt`. The `layout` option specifies how partitions on the device are to be arranged. If `layout` is set to `true`, a single partition using all the space on the device will be created. If set to `false`, no partitions will be created. Partitions can be specified by providing a list to `layout`, where each entry in the list is either a size or a list containing a size and the numerical value for a partition type. The size for partitions is specified in **percentage** of disk space, not in bytes (e.g. a size of 33 would take up 1/3 of the disk space). The `overwrite` option controls whether this module tries to be safe about writing partition talbes or not. If `overwrite: false` is set, the device will be checked for a partition table and for a file system and if either is found, the operation will be skipped. If `overwrite: true` is set, no checks will be performed.

Note: Using `overwrite: true` is dangerous and can lead to data loss, so double check that the correct device has been specified if using this option.

File system configuration is done using the `fs_setup` directive. This config directive accepts a list of filesystem configs. The device to create the filesystem on may be specified either as a path or as an alias in the format `<alias name>.<y>` where `<y>` denotes the partition number on the device. The partition can also be specified by setting `partition` to the desired partition number. The `partition` option may also be set to `auto`, in which this module will search for the existance of a filesystem matching the `label`, `type` and `device` of the `fs_setup` entry and will skip creating the filesystem if one is found. The `partition` option may also be set to `any`, in which case any file system that matches `type` and `device` will cause this module to skip filesystem creation for the `fs_setup` entry, regardless of `label` matching or not. To write a filesystem directly to a device, use `partition: none`. A label can be specified for the filesystem using `label`, and the filesystem type can be specified using `filesystem`.

Note: If specifying device using the `<device name>.<partition number>` format, the value of `partition` will be overwritten.

Note: Using `overwrite: true` for filesystems is dangerous and can lead to data loss, so double check the entry in `fs_setup`.

Note: `replace_fs` is ignored unless `partition` is `auto` or `any`.

Internal name: `cc_disk_setup`

Module frequency: per instance

Supported distros: all

Config keys:

```
device_aliases:
    <alias name>: <device path>
disk_setup:
    <alias name/path>:
        table_type: <'mbr'/'gpt'>
        layout:
            - [33,82]
            - 66
        overwrite: <true/false>
fs_setup:
    - label: <label>
      filesystem: <filesystem type>
      device: <device>
      partition: <"auto"/"any"/"none"/<partition number>>
      overwrite: <true/false>
      replace_fs: <filesystem type>
```

Emit Upstart

Summary: emit upstart configuration

Emit upstart configuration for cloud-init modules on upstart based systems. No user configuration should be required.

Internal name: `cc_emit_upstart`

Module frequency: per always

Supported distros: ubuntu, debian

Fan

Summary: configure ubuntu fan networking

This module installs, configures and starts the ubuntu fan network system. For more information about Ubuntu Fan, see: `https://wiki.ubuntu.com/FanNetworking`.

If cloud-init sees a `fan` entry in cloud-config it will:

- write `config_path` with the contents of the `config` key
- install the package `ubuntu-fan` if it is not installed
- ensure the service is started (or restarted if was previously running)

Internal name: `cc_fan`

Module frequency: per instance

Supported distros: ubuntu

Config keys:

```
fan:
    config: |
        # fan 240
        10.0.0.0/8 eth0/16 dhcp
        10.0.0.0/8 eth1/16 dhcp off
        # fan 241
        241.0.0.0/8 eth0/16 dhcp
    config_path: /etc/network/fan
```

Final Message

Summary: output final message when cloud-init has finished

This module configures the final message that cloud-init writes. The message is specified as a jinja template with the following variables set:

- `version`: cloud-init version
- `timestamp`: time at cloud-init finish
- `datasource`: cloud-init data source
- `uptime`: system uptime

Internal name: `cc_final_message`

Module frequency: per always

Supported distros: all

Config keys:

```
final_message: <message>
```

Foo

Summary: example module

Example to show module structure. Does not do anything.

Internal name: `cc_foo`

Module frequency: per instance

Supported distros: all

Growpart

Summary: grow partitions

Growpart resizes partitions to fill the available disk space. This is useful for cloud instances with a larger amount of disk space available than the pristine image uses, as it allows the instance to automatically make use of the extra space.

The devices run growpart on are specified as a list under the `devices` key. Each entry in the devices list can be either the path to the device's mountpoint in the filesystem or a path to the block device in `/dev`.

The utility to use for resizing can be selected using the `mode` config key. If `mode` key is set to `auto`, then any available utility (either `growpart` or `gpart`) will be used. If neither utility is available, no error will be raised. If `mode` is set to `growpart`, then the `growpart` utility will be used. If this utility is not available on the system, this will result in an error. If `mode` is set to `off` or `false`, then `cc_growpart` will take no action.

There is some functionality overlap between this module and the `growroot` functionality of `cloud-initramfs-tools`. However, there are some situations where one tool is able to function and the other is not. The default configuration for both should work for most cloud instances. To explicitly prevent `cloud-initramfs-tools` from running `growroot`, the file `/etc/growroot-disabled` can be created. By default, both `growroot` and `cc_growpart` will check for the existance of this file and will not run if it is present. However, this file can be ignored for `cc_growpart` by setting `ignore_growroot_disabled` to `true`. For more information on `cloud-initramfs-tools` see: https://launchpad.net/cloud-initramfs-tools

Growpart is enabled by default on the root partition. The default config for growpart is:

```
growpart:
    mode: auto
    devices: ["/"]
    ignore_growroot_disabled: false
```

Internal name: `cc_growpart`

Module frequency: per always

Supported distros: all

Config keys:

```
growpart:
    mode: <auto/growpart/off/false>
    devices:
        - "/"
        - "/dev/vdb1"
    ignore_growroot_disabled: <true/false>
```

Grub Dpkg

Summary: configure grub debconf installation device

Configure which device is used as the target for grub installation. This module should work correctly by default without any user configuration. It can be enabled/disabled using the `enabled` config key in the `grub_dpkg` config dict. The global config key `grub-dpkg` is an alias for `grub_dpkg`. If no installation device is specified this module will look for the first existing device in:

- `/dev/sda`
- `/dev/vda`
- `/dev/xvda`

- /dev/sda1
- /dev/vda1
- /dev/xvda1

Internal name: cc_grub_dpkg

Module frequency: per instance

Supported distros: ubuntu, debian

Config keys:

```
grub_dpkg:
    enabled: <true/false>
    grub-pc/install_devices: <devices>
    grub-pc/install_devices_empty: <devices>
grub-dpkg: (alias for grub_dpkg)
```

Keys to Console

Summary: control which ssh keys may be written to console

For security reasons it may be desirable not to write ssh fingerprints and keys to the console. To avoid the fingerprint of types of ssh keys being written to console the ssh_fp_console_blacklist config key can be used. By default all types of keys will have their fingerprints written to console. To avoid keys of a key type being written to console the ssh_key_console_blacklist config key can be used. By default ssh-dss keys are not written to console.

Internal name: cc_keys_to_console

Module frequency: per instance

Supported distros: all

Config keys:

```
ssh_fp_console_blacklist: <list of key types>
ssh_key_console_blacklist: <list of key types>
```

Landscape

Summary: install and configure landscape client

This module installs and configures landscape-client. The landscape client will only be installed if the key landscape is present in config. Landscape client configuration is given under the client key under the main landscape config key. The config parameters are not interpreted by cloud-init, but rather are converted into a ConfigObj formatted file and written out to /etc/landscape/client.conf.

The following default client config is provided, but can be overridden:

```
landscape:
    client:
        log_level: "info"
        url: "https://landscape.canonical.com/message-system"
        ping_url: "http://landscape.canoncial.com/ping"
        data_path: "/var/lib/landscape/client"
```

Note: see landscape documentation for client config keys

Note: if `tags` is defined, its contents should be a string delimited with `,` rather than a list

Internal name: `cc_landscape`

Module frequency: per instance

Supported distros: ubuntu

Config keys:

```
landscape:
    client:
        url: "https://landscape.canonical.com/message-system"
        ping_url: "http://landscape.canonical.com/ping"
        data_path: "/var/lib/landscape/client"
        http_proxy: "http://my.proxy.com/foobar"
        https_proxy: "https://my.proxy.com/foobar"
        tags: "server,cloud"
        computer_title: "footitle"
        registration_key: "fookey"
        account_name: "fooaccount"
```

Locale

Summary: set system locale

Configure the system locale and apply it system wide. By default use the locale specified by the datasource.

Internal name: `cc_locale`

Module frequency: per instance

Supported distros: all

Config keys:

```
locale: <locale str>
locale_configfile: <path to locale config file>
```

LXD

Summary: configure lxd with `lxd init` and optionally lxd-bridge

This module configures lxd with user specified options using `lxd init`. If lxd is not present on the system but lxd configuration is provided, then lxd will be installed. If the selected storage backend is zfs, then zfs will be installed if missing. If network bridge configuration is provided, then lxd-bridge will be configured accordingly.

Internal name: `cc_lxd`

Module frequency: per instance

Supported distros: ubuntu

Config keys:

```
lxd:
    init:
        network_address: <ip addr>
        network_port: <port>
        storage_backend: <zfs/dir>
        storage_create_device: <dev>
        storage_create_loop: <size>
        storage_pool: <name>
        trust_password: <password>
    bridge:
        mode: <new, existing or none>
        name: <name>
        ipv4_address: <ip addr>
        ipv4_netmask: <cidr>
        ipv4_dhcp_first: <ip addr>
        ipv4_dhcp_last: <ip addr>
        ipv4_dhcp_leases: <size>
        ipv4_nat: <bool>
        ipv6_address: <ip addr>
        ipv6_netmask: <cidr>
        ipv6_nat: <bool>
        domain: <domain>
```

Mcollective

Summary: install, configure and start mcollective

This module installs, configures and starts mcollective. If the `mcollective` key is present in config, then mcollective will be installed and started.

Configuration for `mcollective` can be specified in the `conf` key under `mcollective`. Each config value consists of a key value pair and will be written to `/etc/mcollective/server.cfg`. The `public-cert` and `private-cert` keys, if present in conf may be used to specify the public and private certificates for mcollective. Their values will be written to `/etc/mcollective/ssl/server-public.pem` and `/etc/mcollective/ssl/server-private.pem`.

Note: The ec2 metadata service is readable by non-root users. If security is a concern, use include-once and ssl urls.

Internal name: `cc_mcollective`

Module frequency: per instance

Supported distros: all

Config keys:

```
mcollective:
    conf:
        <key>: <value>
        public-cert: |
            -------BEGIN CERTIFICATE--------
            <cert data>
            -------END CERTIFICATE--------
        private-cert: |
            -------BEGIN CERTIFICATE--------
```

```
            <cert data>
            -------END CERTIFICATE--------
```

Migrator

Summary: migrate old versions of cloud-init data to new

This module handles moving old versions of cloud-init data to newer ones. Currently, it only handles renaming cloud-init's per-frequency semaphore files to canonicalized name and renaming legacy semaphore names to newer ones. This module is enabled by default, but can be disabled by specifying `migrate: false` in config.

Internal name: `cc_migrator`

Module frequency: per always

Supported distros: all

Config keys:

```
migrate: <true/false>
```

Mounts

Summary: configure mount points and swap files

This module can add or remove mountpoints from `/etc/fstab` as well as configure swap. The `mounts` config key takes a list of fstab entries to add. Each entry is specified as a list of `[ fs_spec, fs_file, fs_vfstype, fs_mntops, fs-freq, fs_passno ]`. For more information on these options, consult the manual for `/etc/fstab`. When specifying the `fs_spec`, if the device name starts with one of `xvd`, `sd`, `hd`, or `vd`, the leading `/dev` may be omitted.

In order to remove a previously listed mount, an entry can be added to the mounts list containing `fs_spec` for the device to be removed but no mountpoint (i.e. `[ sda1 ]` or `[ sda1, null ]`).

The `mount_default_fields` config key allows default options to be specified for the values in a `mounts` entry that are not specified, aside from the `fs_spec` and the `fs_file`. If specified, this must be a list containing 7 values. It defaults to:

```
mount_default_fields: [none, none, "auto", "defaults,nobootwait", "0", "2"]
```

On a systemd booted system that default is the mostly equivalent:

```
mount_default_fields: [none, none, "auto",
   "defaults,nofail,x-systemd.requires=cloud-init.service", "0", "2"]
```

Note that *nobootwait* is an upstart specific boot option that somewhat equates to the more standard *nofail*.

Swap files can be configured by setting the path to the swap file to create with `filename`, the size of the swap file with `size` maximum size of the swap file if using an `size: auto` with `maxsize`. By default no swap file is created.

Internal name: `cc_mounts`

Module frequency: per instance

Supported distros: all

Config keys:

```
mounts:
    - [ /dev/ephemeral0, /mnt, auto, "defaults,noexec" ]
    - [ sdc, /opt/data ]
    - [ xvdh, /opt/data, "auto", "defaults,nofail", "0", "0" ]
mount_default_fields: [None, None, "auto", "defaults,nofail", "0", "2"]
swap:
    filename: <file>
    size: <"auto"/size in bytes>
    maxsize: <size in bytes>
```

NTP

Summary: enable and configure ntp

Handle ntp configuration. If ntp is not installed on the system and ntp configuration is specified, ntp will be installed. If there is a default ntp config file in the image or one is present in the distro's ntp package, it will be copied to `/etc/ntp.conf.dist` before any changes are made. A list of ntp pools and ntp servers can be provided under the `ntp` config key. If no ntp `servers` or `pools` are provided, 4 pools will be used in the format `{0-3}.{distro}.pool.ntp.org`.

Internal name: `cc_ntp`

Module frequency: once-per-instance

Supported distros: centos, debian, fedora, opensuse, ubuntu

Config schema: **ntp:** (object/null)

> **pools:** (array of string) List of ntp pools. If both pools and servers are empty, 4 default pool servers will be provided of the format `{0-3}.{distro}.pool.ntp.org`.
>
> **servers:** (array of string) List of ntp servers. If both pools and servers are empty, 4 default pool servers will be provided with the format `{0-3}.{distro}.pool.ntp.org`.

Examples:

```
ntp:
  pools: [0.int.pool.ntp.org, 1.int.pool.ntp.org, ntp.myorg.org]
  servers:
    - ntp.server.local
    - ntp.ubuntu.com
    - 192.168.23.2
```

Package Update Upgrade Install

Summary: update, upgrade, and install packages

This module allows packages to be updated, upgraded or installed during boot. If any packages are to be installed or an upgrade is to be performed then the package cache will be updated first. If a package installation or upgrade requires a reboot, then a reboot can be performed if `package_reboot_if_required` is specified. A list of packages to install can be provided. Each entry in the list can be either a package name or a list with two entries, the first being the package name and the second being the specific package version to install.

Internal name: `cc_package_update_upgrade_install`

Module frequency: per instance

Supported distros: all

Config keys:

```
packages:
    - pwgen
    - pastebinit
    - [libpython2.7, 2.7.3-0ubuntu3.1]
package_update: <true/false>
package_upgrade: <true/false>
package_reboot_if_required: <true/false>

apt_update: (alias for package_update)
apt_upgrade: (alias for package_upgrade)
apt_reboot_if_required: (alias for package_reboot_if_required)
```

Phone Home

Summary: post data to url

This module can be used to post data to a remote host after boot is complete. If the post url contains the string `$INSTANCE_ID` it will be replaced with the id of the current instance. Either all data can be posted or a list of keys to post. Available keys are:

- `pub_key_dsa`
- `pub_key_rsa`
- `pub_key_ecdsa`
- `instance_id`
- `hostname`
- `fdqn`

Internal name: `cc_phone_home`

Module frequency: per instance

Supported distros: all

Config keys:

```
phone_home:
    url: http://example.com/$INSTANCE_ID/
    post:
        - pub_key_dsa
        - instance_id
        - fqdn
    tries: 10
```

Power State Change

Summary: change power state

This module handles shutdown/reboot after all config modules have been run. By default it will take no action, and the system will keep running unless a package installation/upgrade requires a system reboot (e.g. installing a new kernel) and `package_reboot_if_required` is true. The `power_state` config key accepts a dict of options. If `mode` is any value other than `poweroff`, `halt`, or `reboot`, then no action will be taken.

The system can be shutdown before cloud-init has finished using the `timeout` option. The `delay` key specifies a duration to be added onto any shutdown command used. Therefore, if a 5 minute delay and a 120 second shutdown are specified, the maximum amount of time between cloud-init starting and the system shutting down is 7 minutes, and the minimum amount of time is 5 minutes. The `delay` key must have an argument in a form that the `shutdown` utility recognizes. The most common format is the form `+5` for 5 minutes. See `man shutdown` for more options.

Optionally, a command can be run to determine whether or not the system should shut down. The command to be run should be specified in the `condition` key. For command formatting, see the documentation for `cc_runcmd`. The specified shutdown behavior will only take place if the `condition` key is omitted or the command specified by the `condition` key returns 0.

Internal name: `cc_power_state_change`

Module frequency: per instance

Supported distros: all

Config keys:

```
power_state:
    delay: <now/'+minutes'>
    mode: <poweroff/halt/reboot>
    message: <shutdown message>
    timeout: <seconds>
    condition: <true/false/command>
```

Puppet

Summary: install, configure and start puppet

This module handles puppet installation and configuration. If the `puppet` key does not exist in global configuration, no action will be taken. If a config entry for `puppet` is present, then by default the latest version of puppet will be installed. If `install` is set to `false`, puppet will not be installed. However, this will result in an error if puppet is not already present on the system. The version of puppet to be installed can be specified under `version`, and defaults to `none`, which selects the latest version in the repos. If the `puppet` config key exists in the config archive, this module will attempt to start puppet even if no installation was performed.

Puppet configuration can be specified under the `conf` key. The configuration is specified as a dictionary containing high-level `<section>` keys and lists of `<key>=<value>` pairs within each section. Each section name and `<key>=<value>` pair is written directly to `puppet.conf`. As such, section names should be one of: `main`, `master`, `agent` or `user` and keys should be valid puppet configuration options. The `certname` key supports string substitutions for `%i` and `%f`, corresponding to the instance id and fqdn of the machine respectively. If `ca_cert` is present, it will not be written to `puppet.conf`, but instead will be used as the puppermaster certificate. It should be specified in pem format as a multi-line string (using the `|` yaml notation).

Internal name: `cc_puppet`

Module frequency: per instance

Supported distros: all

Config keys:

```
puppet:
    install: <true/false>
    version: <version>
    conf:
        agent:
            server: "puppetmaster.example.org"
```

```
            certname: "%i.%f"
            ca_cert: |
                -------BEGIN CERTIFICATE-------
                <cert data>
                -------END CERTIFICATE-------
```

Resizefs

Summary: Resize filesystem

Resize a filesystem to use all avaliable space on partition. This module is useful along with `cc_growpart` and will ensure that if the root partition has been resized the root filesystem will be resized along with it. By default, `cc_resizefs` will resize the root partition and will block the boot process while the resize command is running. Optionally, the resize operation can be performed in the background while cloud-init continues running modules. This can be enabled by setting `resize_rootfs` to `true`. This module can be disabled altogether by setting `resize_rootfs` to `false`.

Internal name: `cc_resizefs`

Module frequency: always

Supported distros: all

Config schema: **resize_rootfs:** (true/false/noblock) Whether to resize the root partition. Default: 'true'

Examples:

```
resize_rootfs: false  # disable root filesystem resize operation
```

Resolv Conf

Summary: configure resolv.conf

This module is intended to manage resolv.conf in environments where early configuration of resolv.conf is necessary for further bootstrapping and/or where configuration management such as puppet or chef own dns configuration. As Debian/Ubuntu will, by default, utilize resovlconf, and similarly RedHat will use sysconfig, this module is likely to be of little use unless those are configured correctly.

Note: For RedHat with sysconfig, be sure to set PEERDNS=no for all DHCP enabled NICs.

Note: And, in Ubuntu/Debian it is recommended that DNS be configured via the standard /etc/network/interfaces configuration file.

Internal name: `cc_resolv_conf`

Module frequency: per instance

Supported distros: fedora, rhel, sles

Config keys:

```
manage_resolv_conf: <true/false>
resolv_conf:
    nameservers: ['8.8.4.4', '8.8.8.8']
```

```
    searchdomains:
        - foo.example.com
        - bar.example.com
    domain: example.com
    options:
        rotate: <true/false>
        timeout: 1
```

RedHat Subscription

Summary: register red hat enterprise linux based system

Register a RedHat system either by username and password *or* activation and org. Following a sucessful registration, you can auto-attach subscriptions, set the service level, add subscriptions based on pool id, enable/disable yum repositories based on repo id, and alter the rhsm_baseurl and server-hostname in `/etc/rhsm/rhs.conf`. For more details, see the `Register RedHat Subscription` example config.

Internal name: `cc_rh_subscription`

Module frequency: per instance

Supported distros: rhel, fedora

Config keys:

```
rh_subscription:
    username: <username>
    password: <password>
    activation-key: <activation key>
    org: <org number>
    auto-attach: <true/false>
    service-level: <service level>
    add-pool: <list of pool ids>
    enable-repo: <list of yum repo ids>
    disable-repo: <list of yum repo ids>
    rhsm-baseurl: <url>
    server-hostname: <hostname>
```

Rightscale Userdata

Summary: support rightscale configuration hooks

This module adds support for RightScale configuration hooks to cloud-init. RightScale adds a entry in the format `CLOUD_INIT_REMOTE_HOOK=http://...` to ec2 user-data. This module checks for this line in the raw userdata and retrieves any scripts linked by the RightScale user data and places them in the user scripts configuration directory, to be run later by `cc_scripts_user`.

Note: the `CLOUD_INIT_REMOTE_HOOK` config variable is present in the raw ec2 user data only, not in any cloud-config parts

---

Internal name: `cc_rightscale_userdata`

Module frequency: per instance

Supported distros: all

Config keys:

```
CLOUD_INIT_REMOTE_HOOK=<url>
```

Rsyslog

Summary: configure system loggig via rsyslog

This module configures remote system logging using rsyslog.

The rsyslog config file to write to can be specified in `config_filename`, which defaults to `20-cloud-config.conf`. The rsyslog config directory to write config files to may be specified in `config_dir`, which defaults to `/etc/rsyslog.d`.

A list of configurations for for rsyslog can be specified under the `configs` key in the `rsyslog` config. Each entry in `configs` is either a string or a dictionary. Each config entry contains a configuration string and a file to write it to. For config entries that are a dictionary, `filename` sets the target filename and `content` specifies the config string to write. For config entries that are only a string, the string is used as the config string to write. If the filename to write the config to is not specified, the value of the `config_filename` key is used. A file with the selected filename will be written inside the directory specified by `config_dir`.

The command to use to reload the rsyslog service after the config has been updated can be specified in `service_reload_command`. If this is set to `auto`, then an appropriate command for the distro will be used. This is the default behavior. To manually set the command, use a list of command args (e.g. `[systemctl, restart, rsyslog]`).

Configuration for remote servers can be specified in `configs`, but for convenience it can be specified as key value pairs in `remotes`. Each key is the name for an rsyslog remote entry. Each value holds the contents of the remote config for rsyslog. The config consists of the following parts:

- filter for log messages (defaults to `*.*`)
- optional leading `@` or `@@`, indicating udp and tcp respectively (defaults to `@`, for udp)
- ipv4 or ipv6 hostname or address. ipv6 addresses must be in `[::1]` format, (e.g. `@[fd00::1]:514`)
- optional port number (defaults to `514`)

This module will provide sane defaults for any part of the remote entry that is not specified, so in most cases remote hosts can be specified just using `<name>: <address>`.

For backwards compatibility, this module still supports legacy names for the config entries. Legacy to new mappings are as follows:

- `rsyslog` -> `rsyslog/configs`
- `rsyslog_filename` -> `rsyslog/config_filename`
- `rsyslog_dir` -> `rsyslog/config_dir`

Note: The legacy config format does not support specifying `service_reload_command`.

Internal name: `cc_rsyslog`

Module frequency: per instance

Supported distros: all

Config keys:

```
rsyslog:
    config_dir: config_dir
    config_filename: config_filename
    configs:
        - "*.* @@192.158.1.1"
        - content: "*.*   @@192.0.2.1:10514"
          filename: 01-example.conf
        - content: |
            *.*   @@syslogd.example.com
    remotes:
        maas: "192.168.1.1"
        juju: "10.0.4.1"
    service_reload_command: [your, syslog, restart, command]
```

Legacy config keys:

```
rsyslog:
    - "*.* @@192.158.1.1"
rsyslog_dir: /etc/rsyslog-config.d/
rsyslog_filename: 99-local.conf
```

Runcmd

Summary: Run arbitrary commands

Run arbitrary commands at a rc.local like level with output to the console. Each item can be either a list or a string. If the item is a list, it will be properly executed as if passed to `execve()` (with the first arg as the command). If the item is a string, it will be written to a file and interpreted using `sh`.

all commands must be proper yaml, so you have to quote any characters yaml would eat (':' can be problematic)

Internal name: `cc_runcmd`

Module frequency: once-per-instance

Supported distros: all

Config schema: **runcmd:** (array of (array of string)/(string))

Examples:

```
runcmd:
    - [ ls, -l, / ]
    - [ sh, -xc, "echo $(date) ': hello world!'" ]
    - [ sh, -c, echo "=========hello world'=========" ]
    - ls -l /root
    - [ wget, "http://example.org", -O, /tmp/index.html ]
```

Salt Minion

Summary: set up and run salt minion

This module installs, configures and starts salt minion. If the `salt_minion` key is present in the config parts, then salt minion will be installed and started. Configuration for salt minion can be specified in the `conf` key under `salt_minion`. Any conf values present there will be assigned in `/etc/salt/minion`. The public and private keys to use for salt minion can be specified with `public_key` and `private_key` respectively.

Internal name: `cc_salt_minion`

Module frequency: per instance

Supported distros: all

Config keys:

```
salt_minion:
    conf:
        master: salt.example.com
    public_key: |
        ------BEGIN PUBLIC KEY-------
        <key data>
        ------END PUBLIC KEY-------
    private_key: |
        ------BEGIN PRIVATE KEY------
        <key data>
        ------END PRIVATE KEY-------
```

Scripts Per Boot

Summary: run per boot scripts

Any scripts in the `scripts/per-boot` directory on the datasource will be run every time the system boots. Scripts will be run in alphabetical order. This module does not accept any config keys.

Internal name: `cc_scripts_per_boot`

Module frequency: per always

Supported distros: all

Scripts Per Instance

Summary: run per instance scripts

Any scripts in the `scripts/per-instance` directory on the datasource will be run when a new instance is first booted. Scripts will be run in alphabetical order. This module does not accept any config keys.

Internal name: `cc_scripts_per_instance`

Module frequency: per instance

Supported distros: all

Scripts Per Once

Summary: run one time scripts

Any scripts in the `scripts/per-once` directory on the datasource will be run only once. Scripts will be run in alphabetical order. This module does not accept any config keys.

Internal name: `cc_scripts_per_once`

Module frequency: per once

Supported distros: all

Scripts User

Summary: run user scripts

This module runs all user scripts. User scripts are not specified in the `scripts` directory in the datasource, but rather are present in the `scripts` dir in the instance configuration. Any cloud-config parts with a `#!` will be treated as a script and run. Scripts specified as cloud-config parts will be run in the order they are specified in the configuration. This module does not accept any config keys.

Internal name: `cc_scripts_user`

Module frequency: per instance

Supported distros: all

Scripts Vendor

Summary: run vendor scripts

Any scripts in the `scripts/vendor` directory in the datasource will be run when a new instance is first booted. Scripts will be run in alphabetical order. Vendor scripts can be run with an optional prefix specified in the `prefix` entry under the `vendor_data` config key.

Internal name: `cc_scripts_vendor`

Module frequency: per instance

Supported distros: all

Config keys:

```
vendor_data:
    prefix: <vendor data prefix>
```

Seed Random

Summary: provide random seed data

Since all cloud instances started from the same image will produce very similar data when they are first booted, as they are all starting with the same seed for the kernel's entropy keyring. To avoid this, random seed data can be provided to the instance either as a string or by specifying a command to run to generate the data.

Configuration for this module is under the `random_seed` config key. The `file` key specifies the path to write the data to, defaulting to `/dev/urandom`. Data can be passed in directly with `data`, and may optionally be specified in encoded form, with the encoding specified in `encoding`.

Note: when using a multiline value for `data` or specifying binary data, be sure to follow yaml syntax and use the `|` and `!binary` yaml format specifiers when appropriate

---

Instead of specifying a data string, a command can be run to generate/collect the data to be written. The command should be specified as a list of args in the `command` key. If a command is specified that cannot be run, no error will be reported unless `command_required` is set to true.

For example, to use `pollinate` to gather data from a remote entropy server and write it to `/dev/urandom`, the following could be used:

```
random_seed:
    file: /dev/urandom
    command: ["pollinate", "--server=http://local.polinate.server"]
    command_required: true
```

Internal name: cc_seed_random

Module frequency: per instance

Supported distros: all

Config keys:

```
random_seed:
    file: <file>
    data: <random string>
    encoding: <raw/base64/b64/gzip/gz>
    command: [<cmd name>, <arg1>, <arg2>...]
    command_required: <true/false>
```

Set Hostname

Summary: set hostname and fqdn

This module handles setting the system hostname and fqdn. If preserve_hostname is set, then the hostname will not be altered.

A hostname and fqdn can be provided by specifying a full domain name under the fqdn key. Alternatively, a hostname can be specified using the hostname key, and the fqdn of the cloud wil be used. If a fqdn specified with the hostname key, it will be handled properly, although it is better to use the fqdn config key. If both fqdn and hostname are set, fqdn will be used.

Internal name: per instance

Supported distros: all

Config keys:

```
preserve_hostname: <true/false>
fqdn: <fqdn>
hostname: <fqdn/hostname>
```

Set Passwords

Summary: Set user passwords

Set system passwords and enable or disable ssh password authentication. The chpasswd config key accepts a dictionary containing a single one of two keys, either expire or list. If expire is specified and is set to false, then the password global config key is used as the password for all user accounts. If the expire key is specified and is set to true then user passwords will be expired, preventing the default system passwords from being used.

If the list key is provided, a list of username:password pairs can be specified. The usernames specified must already exist on the system, or have been created using the cc_users_groups module. A password can be randomly generated using username:RANDOM or username:R. A hashed password can be specified using username:6salt$hash. Password ssh authentication can be enabled, disabled, or left to system defaults using ssh_pwauth.

Note: if using `expire: true` then a ssh authkey should be specified or it may not be possible to login to the system

Internal name: `cc_set_passwords`

Module frequency: per instance

Supported distros: all

Config keys:

```
ssh_pwauth: <yes/no/unchanged>

password: password1
chpasswd:
    expire: <true/false>

chpasswd:
    list: |
        user1:password1
        user2:RANDOM
        user3:password3
        user4:R

##
# or as yaml list
##
chpasswd:
    list:
        - user1:password1
        - user2:RANDOM
        - user3:password3
        - user4:R
        - user4:$6$rL..$ej...
```

Snappy

Summary: snappy modules allows configuration of snappy.

The below example config config would install `etcd`, and then install `pkg2.smoser` with a `<config-file>` argument where `config-file` has `config-blob` inside it. If `pkgname` is installed already, then `snappy config pkgname <file>` will be called where `file` has `pkgname-config-blob` as its content.

Entries in `config` can be namespaced or non-namespaced for a package. In either case, the config provided to snappy command is non-namespaced. The package name is provided as it appears.

If `packages_dir` has files in it that end in `.snap`, then they are installed. Given 3 files:

- <packages_dir>/foo.snap
- <packages_dir>/foo.config
- <packages_dir>/bar.snap

cloud-init will invoke:

- snappy install <packages_dir>/foo.snap <packages_dir>/foo.config
- snappy install <packages_dir>/bar.snap

Note: that if provided a `config` entry for `ubuntu-core`, then cloud-init will invoke: snappy config ubuntu-core <config> Allowing you to configure ubuntu-core in this way.

The `ssh_enabled` key controls the system's ssh service. The default value is `auto`. Options are:

- **True:** enable ssh service
- **False:** disable ssh service
- **auto:** enable ssh service if either ssh keys have been provided or user has requested password authentication (ssh_pwauth).

Internal name: `cc_snappy`

Module frequency: per instance

Supported distros: ubuntu

Config keys:

```
#cloud-config
snappy:
    system_snappy: auto
    ssh_enabled: auto
    packages: [etcd, pkg2.smoser]
    config:
        pkgname:
            key2: value2
        pkg2:
            key1: value1
    packages_dir: '/writable/user-data/cloud-init/snaps'
```

Snap Config

Summary: snap_config modules allows configuration of snapd.

This module uses the same `snappy` namespace for configuration but acts only only a subset of the configuration.

If `assertions` is set and the user has included a list of assertions then cloud-init will collect the assertions into a single assertion file and invoke `snap ack <path to file with assertions>` which will attempt to load the provided assertions into the snapd assertion database.

If `email` is set, this value is used to create an authorized user for contacting and installing snaps from the Ubuntu Store. This is done by calling `snap create-user` command.

If `known` is set to True, then it is expected the user also included an assertion of type `system-user`. When `snap create-user` is called cloud-init will append '–known' flag which instructs snapd to look for a system-user assertion with the details. If `known` is not set, then `snap create-user` will contact the Ubuntu SSO for validating and importing a system-user for the instance.

Note: If the system is already managed, then cloud-init will not attempt to create a system-user.

Internal name: `cc_snap_config`

Module frequency: per instance

Supported distros: any with 'snapd' available

Config keys:

```
#cloud-config
snappy:
    assertions:
    - |
    <assertion 1>
    - |
    <assertion 2>
    email: user@user.org
    known: true
```

Spacewalk

Summary: install and configure spacewalk

This module installs spacewalk and applies basic configuration. If the `spacewalk` config key is present spacewalk will be installed. The server to connect to after installation must be provided in the `server` in spacewalk configuration. A proxy to connect through and a activation key may optionally be specified.

For more information about spacewalk see: https://fedorahosted.org/spacewalk/

Internal name: `cc_spacewalk`

Module frequency: per instance

Supported distros: redhat, fedora

Config keys:

```
spacewalk:
   server: <url>
   proxy: <proxy host>
   activation_key: <key>
```

SSH

Summary: configure ssh and ssh keys

This module handles most configuration for ssh and ssh keys. Many images have default ssh keys, which can be removed using `ssh_deletekeys`. Since removing default keys is usually the desired behavior this option is enabled by default.

Keys can be added using the `ssh_keys` configuration key. The argument to this config key should be a dictionary entries for the public and private keys of each desired key type. Entries in the `ssh_keys` config dict should have keys in the format `<key type>_private` and `<key type>_public`, e.g. `rsa_private: <key>` and `rsa_public: <key>`. See below for supported key types. Not all key types have to be specified, ones left unspecified will not be used. If this config option is used, then no keys will be generated.

Note: when specifying private keys in cloud-config, care should be taken to ensure that the communication between the data source and the instance is secure

Note: to specify multiline private keys, use yaml multiline syntax

If no keys are specified using `ssh_keys`, then keys will be generated using `ssh-keygen`. By default one public/private pair of each supported key type will be generated. The key types to generate can be specified using the `ssh_genkeytypes` config flag, which accepts a list of key types to use. For each key type for which this module has been instructed to create a keypair, if a key of the same type is already present on the system (i.e. if `ssh_deletekeys` was false), no key will be generated.

Supported key types for the `ssh_keys` and the `ssh_genkeytypes` config flags are:

- rsa
- dsa
- ecdsa
- ed25519

Root login can be enabled/disabled using the `disable_root` config key. Root login options can be manually specified with `disable_root_opts`. If `disable_root_opts` is specified and contains the string `$USER`, it will be replaced with the username of the default user. By default, root login is disabled, and root login opts are set to:

```
no-port-forwarding,no-agent-forwarding,no-X11-forwarding
```

Authorized keys for the default user/first user defined in `users` can be specified using *ssh_authorized_keys'*. Keys should be specified as a list of public keys.

Note: see the `cc_set_passwords` module documentation to enable/disable ssh password authentication

---

Internal name: `cc_ssh`

Module frequency: per instance

Supported distros: all

Config keys:

```
ssh_deletekeys: <true/false>
ssh_keys:
    rsa_private: |
        -----BEGIN RSA PRIVATE KEY-----
        MIIBxwIBAAJhAKD0YSHy73nUgysO13XsJmd4fHiFyQ+00R7VVu2iV9Qco
        ...
        -----END RSA PRIVATE KEY-----
    rsa_public: ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAGEAoPRhIfLvedSDKw7Xd ...
    dsa_private: |
        -----BEGIN DSA PRIVATE KEY-----
        MIIBxwIBAAJhAKD0YSHy73nUgysO13XsJmd4fHiFyQ+00R7VVu2iV9Qco
        ...
        -----END DSA PRIVATE KEY-----
    dsa_public: ssh-dsa AAAAB3NzaC1yc2EAAAABIwAAAGEAoPRhIfLvedSDKw7Xd ...
ssh_genkeytypes: <key type>
disable_root: <true/false>
disable_root_opts: <disable root options string>
ssh_authorized_keys:
    - ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAGEA3FSyQwBI6Z+nCSjUU ...
    - ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEA3I7VUf2l5gSn5uavROsc5HRDpZ ...
```

SSH Authkey Fingerprints

Summary: log fingerprints of user ssh keys

Write fingerprints of authorized keys for each user to log. This is enabled by default, but can be disabled using `no_ssh_fingerprints`. The hash type for the keys can be specified, but defaults to `md5`.

Internal name: ‘‘ cc_ssh_authkey_fingerprints‘‘

Module frequency: per instance

Supported distros: all

Config keys:

```
no_ssh_fingerprints: <true/false>
authkey_hash: <hash type>
```

SSH Import Id

Summary: import ssh id

This module imports ssh keys from either a public keyserver, usually launchpad or github using `ssh-import-id`. Keys are referenced by the username they are associated with on the keyserver. The keyserver can be specified by prepending either `lp:` for launchpad or `gh:` for github to the username.

Internal name: `cc_ssh_import_id`

Module frequency: per instance

Supported distros: ubuntu, debian

Config keys:

```
ssh_import_id:
    - user
    - gh:user
    - lp:user
```

Timezone

Summary: set system timezone

Set the system timezone. If any args are passed to the module then the first will be used for the timezone. Otherwise, the module will attempt to retrieve the timezone from cloud config.

Internal name: `cc_timezone`

Module frequency: per instance

Supported distros: all

Config keys:

```
timezone: <timezone>
```

Update Etc Hosts

Summary: update `/etc/hosts`

This module will update the contents of `/etc/hosts` based on the hostname/fqdn specified in config. Management of `/etc/hosts` is controlled using `manage_etc_hosts`. If this is set to false, cloud-init will not manage `/etc/hosts` at all. This is the default behavior.

If set to `true` or `template`, cloud-init will generate `/etc/hosts` using the template located in `/etc/cloud/templates/hosts.tmpl`. In the `/etc/cloud/templates/hosts.tmpl` template, the strings `$hostname` and `$fqdn` will be replaced with the hostname and fqdn respectively.

If `manage_etc_hosts` is set to `localhost`, then cloud-init will not rewrite `/etc/hosts` entirely, but rather will ensure that a entry for the fqdn with ip `127.0.1.1` is present in `/etc/hosts` (i.e. `ping <hostname>` will ping `127.0.1.1`).

Note: if `manage_etc_hosts` is set `true` or `template`, the contents of `/etc/hosts` will be updated every boot. to make any changes to `/etc/hosts` persistant they must be made in `/etc/cloud/templates/hosts.tmpl`

Note: for instructions on specifying hostname and fqdn, see documentation for `cc_set_hostname`

Internal name: `cc_update_etc_hosts`

Module frequency: per always

Supported distros: all

Config keys:

```
manage_etc_hosts: <true/"template"/false/"localhost">
fqdn: <fqdn>
hostname: <fqdn/hostname>
```

Update Hostname

Summary: update hostname and fqdn

This module will update the system hostname and fqdn. If `preserve_hostname` is set, then the hostname will not be altered.

Note: for instructions on specifying hostname and fqdn, see documentation for `cc_set_hostname`

Internal name: `cc_update_hostname`

Module frequency: per always

Supported distros: all

Config keys:

```
preserve_hostname: <true/false>
fqdn: <fqdn>
hostname: <fqdn/hostname>
```

Users and Groups

Summary: configure users and groups

This module configures users and groups. For more detailed information on user options, see the `Including users and groups` config example.

Groups to add to the system can be specified as a list under the `groups` key. Each entry in the list should either contain a the group name as a string, or a dictionary with the group name as the key and a list of users who should be members of the group as the value.

The `users` config key takes a list of users to configure. The first entry in this list is used as the default user for the system. To preserve the standard default user for the distro, the string `default` may be used as the first entry of the `users` list. Each entry in the `users` list, other than a `default` entry, should be a dictionary of options for the user. Supported config keys for an entry in `users` are as follows:

- `name`: The user's login name
- `expiredate`: Optional. Date on which the user's login will be disabled. Default: none
- `gecos`: Optional. Comment about the user, usually a comma-separated string of real name and contact information. Default: none
- `groups`: Optional. Additional groups to add the user to. Default: none
- `homedir`: Optional. Home dir for user. Default is `/home/<username>`
- `inactive`: Optional. Mark user inactive. Default: false
- `lock_passwd`: Optional. Disable password login. Default: true
- `no-create-home`: Optional. Do not create home directory. Default: false
- `no-log-init`: Optional. Do not initialize lastlog and faillog for user. Default: false
- `no-user-group`: Optional. Do not create group named after user. Default: false
- `passwd`: Hash of user password
- `primary-group`: Optional. Primary group for user. Default to new group named after user.
- `selinux-user`: Optional. SELinux user for user's login. Default to default SELinux user.
- `shell`: Optional. The user's login shell. The default is to set no shell, which results in a system-specific default being used.
- `snapuser`: Optional. Specify an email address to create the user as a Snappy user through `snap create-user`. If an Ubuntu SSO account is associated with the address, username and SSH keys will be requested from there. Default: none
- `ssh-authorized-keys`: Optional. List of ssh keys to add to user's authkeys file. Default: none
- `ssh-import-id`: Optional. SSH id to import for user. Default: none
- `sudo`: Optional. Sudo rule to use, or list of sudo rules to use. Default: none.
- `system`: Optional. Create user as system user with no home directory. Default: false
- `uid`: Optional. The user's ID. Default: The next available value.

Note: Specifying a hash of a user's password with `passwd` is a security risk if the cloud-config can be intercepted. SSH authentication is preferred.

Note: If specifying a sudo rule for a user, ensure that the syntax for the rule is valid, as it is not checked by cloud-init.

Internal name: `cc_users_groups`

Module frequency: per instance

Supported distros: all

Config keys:

```
groups:
    - <group>: [<user>, <user>]
    - <group>

users:
    - default
    - name: <username>
      expiredate: <date>
      gecos: <comment>
      groups: <additional groups>
      homedir: <home directory>
      inactive: <true/false>
      lock_passwd: <true/false>
      no-create-home: <true/false>
      no-log-init: <true/false>
      no-user-group: <true/false>
      passwd: <password>
      primary-group: <primary group>
      selinux-user: <selinux username>
      shell: <shell path>
      snapuser: <email>
      ssh-authorized-keys:
          - <key>
          - <key>
      ssh-import-id: <id>
      sudo: <sudo config>
      system: <true/false>
      uid: <user id>
```

Write Files

Summary: write arbitrary files

Write out arbitrary content to files, optionally setting permissions. Content can be specified in plain text or binary. Data encoded with either base64 or binary gzip data can be specified and will be decoded before being written.

Note: if multiline data is provided, care should be taken to ensure that it follows yaml formatting standargs. to specify binary data, use the yaml option `!!binary`

Internal name: `cc_write_files`

Module frequency: per instance

Supported distros: all

Config keys:

```
write_files:
    - encoding: b64
      content: CiMgVGhpcyBmaWxlIGNvbnRyb2xzIHRoZSBzdGF0ZSBvZiBTRUxpbnV4...
      owner: root:root
      path: /etc/sysconfig/selinux
      permissions: '0644'
    - content: |
        # My new /etc/sysconfig/samba file

        SMDBOPTIONS="-D"
      path: /etc/sysconfig/samba
    - content: !!binary |
        f0VMRgIBAQAAAAAAAAAAAAIAPgABAAAAwARAAAAAAABAAAAAAAAAAJAVAAAAAA
        AEAAHgAdAAYAAAAFAAAAQAAAAAAAAABAAEAAAAAAAEAAQAAAAAAAwAEAAAAAA
        AAAAAAAAAwAAAAQAAAAAAgAAAAAAAAACQAAAAAAAAAJAAAAAAAAAcAAAAAAAAB
        ...
      path: /bin/arch
      permissions: '0555'
```

Yum Add Repo

Summary: add yum repository configuration to the system

Add yum repository configuration to `/etc/yum.repos.d`. Configuration files are named based on the dictionary key under the `yum_repos` they are specified with. If a config file already exists with the same name as a config entry, the config entry will be skipped.

Internal name: `cc_yum_add_repo`

Module frequency: per always

Supported distros: fedora, rhel

Config keys:

```
yum_repos:
    <repo-name>:
        baseurl: <repo url>
        name: <repo name>
        enabled: <true/false>
        # any repository configuration options (see man yum.conf)
```

Merging User-Data Sections

Overview

This was implemented because it has been a common feature request that there be a way to specify how cloud-config yaml "dictionaries" provided as user-data are merged together when there are multiple yaml files to merge together (say when performing an #include).

Since previously the merging algorithm was very simple and would only overwrite and not append lists, or strings, and so on it was decided to create a new and improved way to merge dictionaries (and their contained objects) together in a way that is customizable, thus allowing for users who provide cloud-config user-data to determine exactly how their objects will be merged.

For example.

```
#cloud-config (1)
run_cmd:
  - bash1
  - bash2

#cloud-config (2)
run_cmd:
  - bash3
  - bash4
```

The previous way of merging the two objects above would result in a final cloud-config object that contains the following.

```
#cloud-config (merged)
run_cmd:
  - bash3
  - bash4
```

Typically this is not what users want; instead they would likely prefer:

```
#cloud-config (merged)
run_cmd:
  - bash1
  - bash2
  - bash3
  - bash4
```

This way makes it easier to combine the various cloud-config objects you have into a more useful list, thus reducing duplication necessary to accomplish the same result with the previous method.

Customizability

Because the above merging algorithm may not always be desired (just as the previous merging algorithm was not always the preferred one), the concept of customized merging was introduced through 'merge classes'.

A merge class is a class definition which provides functions that can be used to merge a given type with another given type.

An example of one of these merging classes is the following:

```
class Merger(object):
    def __init__(self, merger, opts):
        self._merger = merger
        self._overwrite = 'overwrite' in opts

    # This merging algorithm will attempt to merge with
    # another dictionary, on encountering any other type of object
    # it will not merge with said object, but will instead return
    # the original value
    #
    # On encountering a dictionary, it will create a new dictionary
    # composed of the original and the one to merge with, if 'overwrite'
    # is enabled then keys that exist in the original will be overwritten
    # by keys in the one to merge with (and associated values). Otherwise
    # if not in overwrite mode the 2 conflicting keys themselves will
    # be merged.
```

```
    def _on_dict(self, value, merge_with):
        if not isinstance(merge_with, (dict)):
            return value
        merged = dict(value)
        for (k, v) in merge_with.items():
            if k in merged:
                if not self._overwrite:
                    merged[k] = self._merger.merge(merged[k], v)
                else:
                    merged[k] = v
            else:
                merged[k] = v
        return merged
```

As you can see there is a '_on_dict' method here that will be given a source value and a value to merge with. The result will be the merged object. This code itself is called by another merging class which 'directs' the merging to happen by analyzing the types of the objects to merge and attempting to find a know object that will merge that type. I will avoid pasting that here, but it can be found in the *mergers/__init__.py* file (see *LookupMerger* and *UnknownMerger*).

So following the typical cloud-init way of allowing source code to be downloaded and used dynamically, it is possible for users to inject there own merging files to handle specific types of merging as they choose (the basic ones included will handle lists, dicts, and strings). Note how each merge can have options associated with it which affect how the merging is performed, for example a dictionary merger can be told to overwrite instead of attempt to merge, or a string merger can be told to append strings instead of discarding other strings to merge with.

How to activate

There are a few ways to activate the merging algorithms, and to customize them for your own usage.

1. The first way involves the usage of MIME messages in cloud-init to specify multipart documents (this is one way in which multiple cloud-config is joined together into a single cloud-config). Two new headers are looked for, both of which can define the way merging is done (the first header to exist wins). These new headers (in lookup order) are 'Merge-Type' and 'X-Merge-Type'. The value should be a string which will satisfy the new merging format definition (see below for this format).
2. The second way is actually specifying the merge-type in the body of the cloud-config dictionary. There are 2 ways to specify this, either as a string or as a dictionary (see format below). The keys that are looked up for this definition are the following (in order), 'merge_how', 'merge_type'.

String format

The string format that is expected is the following.

```
classname1(option1,option2)+classname2(option3,option4)....
```

The class name there will be connected to class names used when looking for the class that can be used to merge and options provided will be given to the class on construction of that class.

For example, the default string that is used when none is provided is the following:

```
list()+dict()+str()
```

Dictionary format

A dictionary can be used when it specifies the same information as the string format (i.e. the second option above), for example:

```
{'merge_how': [{'name': 'list', 'settings': ['extend']},
               {'name': 'dict', 'settings': []},
               {'name': 'str', 'settings': ['append']}]}
```

This would be the equivalent format for default string format but in dictionary form instead of string form.

Specifying multiple types and its effect

Now you may be asking yourself, if I specify a merge-type header or dictionary for every cloud-config that I provide, what exactly happens?

The answer is that when merging, a stack of 'merging classes' is kept, the first one on that stack is the default merging classes, this set of mergers will be used when the first cloud-config is merged with the initial empty cloud-config dictionary. If the cloud-config that was just merged provided a set of merging classes (via the above formats) then those merging classes will be pushed onto the stack. Now if there is a second cloud-config to be merged then the merging classes from the cloud-config before the first will be used (not the default) and so on. This way a cloud-config can decide how it will merge with a cloud-config dictionary coming after it.

Other uses

In addition to being used for merging user-data sections, the default merging algorithm for merging 'conf.d' yaml files (which form an initial yaml config for cloud-init) was also changed to use this mechanism so its full benefits (and customization) can also be used there as well. Other places that used the previous merging are also, similarly, now extensible (metadata merging, for example).

Note, however, that merge algorithms are not used *across* types of configuration. As was the case before merging was implemented, user-data will overwrite conf.d configuration without merging.

Network Configuration

- Default Behavior
- Disabling Network Configuration
- Fallback Networking
- Network Configuration Sources
- Network Configuration Outputs
- Network Output Policy
- Network Configuration Tools
- Examples

Default Behavior

Cloud-init 's searches for network configuration in order of increasing precedence; each item overriding the previous.

Datasource

For example, OpenStack may provide network config in the MetaData Service.

System Config

A `network:` entry in /etc/cloud/cloud.cfg.d/* configuration files.

Kernel Command Line

`ip=` or `network-config=<YAML config string>`

User-data cannot change an instance's network configuration. In the absence of network configuration in any of the above sources , Cloud-init will write out a network configuration that will issue a DHCP request on a "first" network interface.

Disabling Network Configuration

Users may disable Cloud-init 's network configuration capability and rely on other methods, such as embedded configuration or other customizations.

Cloud-init supports the following methods for disabling cloud-init.

Kernel Command Line

Cloud-init will check for a parameter `network-config` and the value is expected to be YAML string in the *Networking Config Version 1* format. The YAML string may optionally be `Base64` encoded, and optionally compressed with `gzip`.

Example disabling kernel command line entry:

```
network-config={config: disabled}
```

cloud config

In the combined cloud-init configuration dictionary.

```
network:
  config: disabled
```

If Cloud-init 's networking config has not been disabled, and no other network information is found, then it will proceed to generate a fallback networking configuration.

Fallback Network Configuration

Cloud-init will attempt to determine which of any attached network devices is most likely to have a connection and then generate a network configuration to issue a DHCP request on that interface.

Cloud-init runs during early boot and does not expect composed network devices (such as Bridges) to be available. Cloud-init does not consider the following interface devices as likely 'first' network interfaces for fallback configuration; they are filtered out from being selected.

- **loopback**: `name=lo`
- **Virtual Ethernet**: `name=veth*`
- **Software Bridges**: `type=bridge`

- **Software VLANs**: `type=vlan`

Cloud-init will prefer network interfaces that indicate they are connected via the Linux `carrier` flag being set. If no interfaces are marked connected, then all unfiltered interfaces are potential connections.

Of the potential interfaces, Cloud-init will attempt to pick the "right" interface given the information it has available.

Finally after selecting the "right" interface, a configuration is generated and applied to the system.

Network Configuration Sources

Cloud-init accepts a number of different network configuration formats in support of different cloud substrates. The Datasource for these clouds in Cloud-init will detect and consume Datasource-specific network configuration formats for use when writing an instance's network configuration.

The following Datasources optionally provide network configuration:

- *Config Drive*
  - OpenStack Metadata Service Network
  - *Network Configuration ENI (Legacy)*
- *Digital Ocean*
  - DigitalOcean JSON metadata
- *NoCloud*
  - *Networking Config Version 1*
  - *Networking Config Version 2*
  - *Network Configuration ENI (Legacy)*
- *OpenNebula*
  - *Network Configuration ENI (Legacy)*
- *OpenStack*
  - *Network Configuration ENI (Legacy)*
  - OpenStack Metadata Service Network
- *SmartOS Datasource*
  - SmartOS JSON Metadata

For more information on network configuration formats

Network Configuration ENI (Legacy)

Cloud-init supports reading and writing network config in the `ENI` format which is consumed by the `ifupdown` tool to parse and apply network configuration.

As an input format this is **legacy**. In cases where ENI format is available and another format is also available, it will prefer to use the other format. This can happen in either *NoCloud* or *OpenStack* datasources.

Please reference existing documentation for the `/etc/network/interfaces(5)` format.

Networking Config Version 1

This network configuration format lets users customize their instance's networking interfaces by assigning subnet configuration, virtual device creation (bonds, bridges, vlans) routes and DNS configuration.

Required elements of a Network Config Version 1 are `config` and `version`.

Cloud-init will read this format from system config. For example the following could be present in `/etc/cloud/cloud.cfg.d/custom-networking.cfg`:

```
network:
  version: 1
  config:
  - type: physical
    name: eth0
    subnets:
      - type: dhcp
```

The *NoCloud* datasource can also provide cloud-init networking configuration in this Format.

Configuration Types

Within the network `config` portion, users include a list of configuration types. The current list of support `type` values are as follows:

- Physical (`physical`)
- Bond (`bond`)
- Bridge (`bridge`)
- VLAN (`vlan`)
- Nameserver (`nameserver`)
- Route (`route`)

Physical, Bond, Bridge and VLAN types may also include IP configuration under the key `subnets`.

- Subnet/IP (`subnets`)

Physical

The `physical` type configuration represents a "physical" network device, typically Ethernet-based. At least one of of these entries is required for external network connectivity. Type `physical` requires only one key: `name`. A `physical` device may contain some or all of the following keys:

name: *<desired device name>*

A devices name must be less than 15 characters. Names exceeding the maximum will be truncated. This is a limitation of the Linux kernel network-device structure.

mac_address: *<MAC Address>*

The MAC Address is a device unique identifier that most Ethernet-based network devices possess. Specifying a MAC Address is optional.

Note: Cloud-init will handle the persistent mapping between a device's `name` and the `mac_address`.

---

mtu: <*MTU SizeBytes*>

The MTU key represents a device's Maximum Transmission Unit, the largest size packet or frame, specified in octets (eight-bit bytes), that can be sent in a packet- or frame-based network. Specifying `mtu` is optional.

Note: The possible supported values of a device's MTU is not available at configuration time. It's possible to specify a value too large or to small for a device and may be ignored by the device.

Physical Example:

```
network:
  version: 1
  config:
    # Simple network adapter
    - type: physical
      name: interface0
      mac_address: 00:11:22:33:44:55
    # Second nic with Jumbo frames
    - type: physical
      name: jumbo0
      mac_address: aa:11:22:33:44:55
      mtu: 9000
    # 10G pair
    - type: physical
      name: gbe0
      mac_address: cd:11:22:33:44:00
    - type: physical
      name: gbe1
      mac_address: cd:11:22:33:44:02
```

Bond

A `bond` type will configure a Linux software Bond with one or more network devices. A `bond` type requires the following keys:

name: <*desired device name*>

A devices name must be less than 15 characters. Names exceeding the maximum will be truncated. This is a limitation of the Linux kernel network-device structure.

mac_address: <*MAC Address*>

When specifying MAC Address on a bond this value will be assigned to the bond device and may be different than the MAC address of any of the underlying bond interfaces. Specifying a MAC Address is optional. If `mac_address` is not present, then the bond will use one of the MAC Address values from one of the bond interfaces.

bond_interfaces: <*List of network device names*>

The `bond_interfaces` key accepts a list of network device `name` values from the configuration. This list may be empty.

params: <*Dictionary of key: value bonding parameter pairs*>

The `params` key in a bond holds a dictionary of bonding parameters. This dictionary may be empty. For more details on what the various bonding parameters mean please read the Linux Kernel Bonding.txt.

Valid `params` keys are:

- `active_slave`: Set bond attribute

- ad_actor_key: Set bond attribute
- ad_actor_sys_prio: Set bond attribute
- ad_actor_system: Set bond attribute
- ad_aggregator: Set bond attribute
- ad_num_ports: Set bond attribute
- ad_partner_key: Set bond attribute
- ad_partner_mac: Set bond attribute
- ad_select: Set bond attribute
- ad_user_port_key: Set bond attribute
- all_slaves_active: Set bond attribute
- arp_all_targets: Set bond attribute
- arp_interval: Set bond attribute
- arp_ip_target: Set bond attribute
- arp_validate: Set bond attribute
- downdelay: Set bond attribute
- fail_over_mac: Set bond attribute
- lacp_rate: Set bond attribute
- lp_interval: Set bond attribute
- miimon: Set bond attribute
- mii_status: Set bond attribute
- min_links: Set bond attribute
- mode: Set bond attribute
- num_grat_arp: Set bond attribute
- num_unsol_na: Set bond attribute
- packets_per_slave: Set bond attribute
- primary: Set bond attribute
- primary_reselect: Set bond attribute
- queue_id: Set bond attribute
- resend_igmp: Set bond attribute
- slaves: Set bond attribute
- tlb_dynamic_lb: Set bond attribute
- updelay: Set bond attribute
- use_carrier: Set bond attribute
- xmit_hash_policy: Set bond attribute

Bond Example:

```
network:
 version: 1
 config:
   # Simple network adapter
   - type: physical
     name: interface0
     mac_address: 00:11:22:33:44:55
   # 10G pair
   - type: physical
     name: gbe0
     mac_address: cd:11:22:33:44:00
   - type: physical
     name: gbe1
     mac_address: cd:11:22:33:44:02
   - type: bond
     name: bond0
     bond_interfaces:
       - gbe0
       - gbe1
     params:
       bond-mode: active-backup
```

Bridge

Type `bridge` requires the following keys:

- `name`: Set the name of the bridge.
- `bridge_interfaces`: Specify the ports of a bridge via their `name`. This list may be empty.
- `params`: A list of bridge params. For more details, please read the bridge-utils-interfaces manpage.

Valid keys are:

- `bridge_ageing`: Set the bridge's ageing value.
- `bridge_bridgeprio`: Set the bridge device network priority.
- `bridge_fd`: Set the bridge's forward delay.
- `bridge_hello`: Set the bridge's hello value.
- `bridge_hw`: Set the bridge's MAC address.
- `bridge_maxage`: Set the bridge's maxage value.
- `bridge_maxwait`: Set how long network scripts should wait for the bridge to be up.
- `bridge_pathcost`: Set the cost of a specific port on the bridge.
- `bridge_portprio`: Set the priority of a specific port on the bridge.
- `bridge_ports`: List of devices that are part of the bridge.
- `bridge_stp`: Set spanning tree protocol on or off.
- `bridge_waitport`: Set amount of time in seconds to wait on specific ports to become available.

Bridge Example:

```
network:
 version: 1
 config:
   # Simple network adapter
   - type: physical
     name: interface0
     mac_address: 00:11:22:33:44:55
   # Second nic with Jumbo frames
   - type: physical
     name: jumbo0
     mac_address: aa:11:22:33:44:55
     mtu: 9000
   - type: bridge
     name: br0
     bridge_interfaces:
       - jumbo0
     params:
       bridge_ageing: 250
       bridge_bridgeprio: 22
       bridge_fd: 1
       bridge_hello: 1
       bridge_maxage: 10
       bridge_maxwait: 0
       bridge_pathcost:
         - jumbo0 75
       bridge_pathprio:
         - jumbo0 28
       bridge_stp: 'off'
       bridge_maxwait:
         - jumbo0 0
```

VLAN

Type `vlan` requires the following keys:

- `name`: Set the name of the VLAN
- `vlan_link`: Specify the underlying link via its `name`.
- `vlan_id`: Specify the VLAN numeric id.

VLAN Example:

```
network:
  version: 1
  config:
    # Physical interfaces.
    - type: physical
      name: eth0
      mac_address: "c0:d6:9f:2c:e8:80"
    # VLAN interface.
    - type: vlan
      name: eth0.101
      vlan_link: eth0
      vlan_id: 101
      mtu: 1500
```

Nameserver

Users can specify a `nameserver` type. Nameserver dictionaries include the following keys:

- `address`: List of IPv4 or IPv6 address of nameservers.
- `search`: List of of hostnames to include in the resolv.conf search path.

Nameserver Example:

```
network:
  version: 1
  config:
    - type: physical
      name: interface0
      mac_address: 00:11:22:33:44:55
      subnets:
         - type: static
           address: 192.168.23.14/27
           gateway: 192.168.23.1
    - type: nameserver:
      address:
        - 192.168.23.2
        - 8.8.8.8
      search:
        - exemplary
```

Route

Users can include static routing information as well. A `route` dictionary has the following keys:

- `destination`: IPv4 network address with CIDR netmask notation.
- `gateway`: IPv4 gateway address with CIDR netmask notation.
- `metric`: Integer which sets the network metric value for this route.

Route Example:

```
network:
  version: 1
  config:
    - type: physical
      name: interface0
      mac_address: 00:11:22:33:44:55
      subnets:
         - type: static
           address: 192.168.23.14/24
           gateway: 192.168.23.1
    - type: route
      destination: 192.168.24.0/24
      gateway: 192.168.24.1
      metric: 3
```

Subnet/IP

For any network device (one of the Config Types) users can define a list of `subnets` which contain ip configuration dictionaries. Multiple subnet entries will create interface alias allowing a single interface to use different ip configurations.

Valid keys for for `subnets` include the following:

- `type`: Specify the subnet type.
- `control`: Specify manual, auto or hotplug. Indicates how the interface will be handled during boot.
- `address`: IPv4 or IPv6 address. It may include CIDR netmask notation.
- `netmask`: IPv4 subnet mask in dotted format or CIDR notation.
- `gateway`: IPv4 address of the default gateway for this subnet.
- `dns_nameserver`: Specify a list of IPv4 dns server IPs to end up in resolv.conf.
- `dns_search`: Specify a list of search paths to be included in resolv.conf.
- `routes`: Specify a list of routes for a given interface

Subnet types are one of the following:

- `dhcp4`: Configure this interface with IPv4 dhcp.
- `dhcp`: Alias for `dhcp4`
- `dhcp6`: Configure this interface with IPv6 dhcp.
- `static`: Configure this interface with a static IPv4.
- `static6`: Configure this interface with a static IPv6 .

When making use of `dhcp` types, no additional configuration is needed in the subnet dictionary.

Subnet DHCP Example:

```
network:
  version: 1
  config:
    - type: physical
      name: interface0
      mac_address: 00:11:22:33:44:55
      subnets:
        - type: dhcp
```

Subnet Static Example:

```
network:
  version: 1
  config:
    - type: physical
      name: interface0
      mac_address: 00:11:22:33:44:55
      subnets:
        - type: static
          address: 192.168.23.14/27
          gateway: 192.168.23.1
          dns_nameservers:
            - 192.168.23.2
            - 8.8.8.8
```

```
          dns_search:
            - exemplary.maas
```

The following will result in an `interface0` using DHCP and `interface0:1` using the static subnet configuration.

Multiple subnet Example:

```
network:
  version: 1
  config:
    - type: physical
      name: interface0
      mac_address: 00:11:22:33:44:55
      subnets:
        - type: dhcp
        - type: static
          address: 192.168.23.14/27
          gateway: 192.168.23.1
          dns_nameservers:
            - 192.168.23.2
            - 8.8.8.8
          dns_search:
            - exemplary
```

Subnet with routes Example:

```
network:
  version: 1
  config:
    - type: physical
      name: interface0
      mac_address: 00:11:22:33:44:55
      subnets:
        - type: dhcp
        - type: static
          address: 10.184.225.122
          netmask: 255.255.255.252
          routes:
            - gateway: 10.184.225.121
              netmask: 255.240.0.0
              network: 10.176.0.0
            - gateway: 10.184.225.121
              netmask: 255.240.0.0
              network: 10.208.0.0
```

Multi-layered configurations

Complex networking sometimes uses layers of configuration. The syntax allows users to build those layers one at a time. All of the virtual network devices supported allow specifying an underlying device by their `name` value.

Bonded VLAN Example:

```
network:
  version: 1
  config:
    # 10G pair
    - type: physical
```

```
      name: gbe0
      mac_address: cd:11:22:33:44:00
    - type: physical
      name: gbe1
      mac_address: cd:11:22:33:44:02
    # Bond.
    - type: bond
      name: bond0
      bond_interfaces:
        - gbe0
        - gbe1
      params:
        bond-mode: 802.3ad
        bond-lacp-rate: fast
    # A Bond VLAN.
    - type: vlan
        name: bond0.200
        vlan_link: bond0
        vlan_id: 200
        subnets:
            - type: dhcp4
```

More Examples

Some more examples to explore the various options available.

Multiple VLAN example:

```
network:
  version: 1
  config:
  - id: eth0
    mac_address: d4:be:d9:a8:49:13
    mtu: 1500
    name: eth0
    subnets:
    - address: 10.245.168.16/21
      dns_nameservers:
      - 10.245.168.2
      gateway: 10.245.168.1
      type: static
    type: physical
  - id: eth1
    mac_address: d4:be:d9:a8:49:15
    mtu: 1500
    name: eth1
    subnets:
    - address: 10.245.188.2/24
      dns_nameservers: []
      type: static
    type: physical
  - id: eth1.2667
    mtu: 1500
    name: eth1.2667
    subnets:
    - address: 10.245.184.2/24
      dns_nameservers: []
```

```
    type: static
  type: vlan
  vlan_id: 2667
  vlan_link: eth1
- id: eth1.2668
  mtu: 1500
  name: eth1.2668
  subnets:
  - address: 10.245.185.1/24
    dns_nameservers: []
    type: static
  type: vlan
  vlan_id: 2668
  vlan_link: eth1
- id: eth1.2669
  mtu: 1500
  name: eth1.2669
  subnets:
  - address: 10.245.186.1/24
    dns_nameservers: []
    type: static
  type: vlan
  vlan_id: 2669
  vlan_link: eth1
- id: eth1.2670
  mtu: 1500
  name: eth1.2670
  subnets:
  - address: 10.245.187.2/24
    dns_nameservers: []
    type: static
  type: vlan
  vlan_id: 2670
  vlan_link: eth1
- address: 10.245.168.2
  search:
  - dellstack
  type: nameserver
```

Networking Config Version 2

Cloud-init's support for Version 2 network config is a subset of the version 2 format defined for the netplan tool. Cloud-init supports both reading and writing of Version 2; the latter support requires a distro with netplan present.

The `network` key has at least two required elements. First it must include `version: 2` and one or more of possible device `types`..

Cloud-init will read this format from system config. For example the following could be present in `/etc/cloud/cloud.cfg.d/custom-networking.cfg`:

network: version: 2 ethernets: []

It may also be provided in other locations including the *NoCloud*, see *Default Behavior* for other places.

Supported device `types` values are as follows:

- Ethernets (`ethernets`)
- Bonds (`bonds`)

- Bridges (`bridges`)
- VLANs (`vlans`)

Each type block contains device definitions as a map where the keys (called "configuration IDs"). Each entry under the `types` may include IP and/or device configuration.

Cloud-init does not current support `wifis` type that is present in native netplan.

Device configuration IDs

The key names below the per-device-type definition maps (like `ethernets:`) are called "ID"s. They must be unique throughout the entire set of configuration files. Their primary purpose is to serve as anchor names for composite devices, for example to enumerate the members of a bridge that is currently being defined.

There are two physically/structurally different classes of device definitions, and the ID field has a different interpretation for each:

Physical devices

: (Examples: ethernet, wifi) These can dynamically come and go between reboots and even during runtime (hotplugging). In the generic case, they can be selected by `match:` rules on desired properties, such as name/name pattern, MAC address, driver, or device paths. In general these will match any number of devices (unless they refer to properties which are unique such as the full path or MAC address), so without further knowledge about the hardware these will always be considered as a group.

It is valid to specify no match rules at all, in which case the ID field is simply the interface name to be matched. This is mostly useful if you want to keep simple cases simple, and it's how network device configuration has been done for a long time.

If there are `match:` rules, then the ID field is a purely opaque name which is only being used for references from definitions of compound devices in the config.

Virtual devices

: (Examples: veth, bridge, bond) These are fully under the control of the config file(s) and the network stack. I. e. these devices are being created instead of matched. Thus `match:` and `set-name:` are not applicable for these, and the ID field is the name of the created virtual device.

Common properties for physical device types

match: *<(mapping)>*

This selects a subset of available physical devices by various hardware properties. The following configuration will then apply to all matching devices, as soon as they appear. *All* specified properties must match. The following properties for creating matches are supported:

name: *<(scalar)>*

Current interface name. Globs are supported, and the primary use case for matching on names, as selecting one fixed name can be more easily achieved with having no `match:` at all and just using the ID (see above). Note that currently only networkd supports globbing, NetworkManager does not.

macaddress: *<(scalar)>*

Device's MAC address in the form "XX:XX:XX:XX:XX:XX". Globs are not allowed.

driver: *<(scalar)>*

Kernel driver name, corresponding to the `DRIVER` udev property. Globs are supported. Matching on driver is *only* supported with networkd.

Examples:

```
# all cards on second PCI bus
match:
  name: enp2*

# fixed MAC address
match:
  macaddress: 11:22:33:AA:BB:FF

# first card of driver ``ixgbe``
match:
  driver: ixgbe
  name: en*s0
```

set-name: *<(scalar)>*

When matching on unique properties such as path or MAC, or with additional assumptions such as "there will only ever be one wifi device", match rules can be written so that they only match one device. Then this property can be used to give that device a more specific/desirable/nicer name than the default from udev's ifnames. Any additional device that satisfies the match rules will then fail to get renamed and keep the original kernel name (and dmesg will show an error).

wakeonlan: *<(bool)>*

Enable wake on LAN. Off by default.

Common properties for all device types

renderer: *<(scalar)>*

Use the given networking backend for this definition. Currently supported are `networkd` and `NetworkManager`. This property can be specified globally in `networks:`, for a device type (in e. g. `ethernets:`) or for a particular device definition. Default is `networkd`.

Note: Cloud-init only supports networkd backend if rendering version2 config to the instance.

---

dhcp4: *<(bool)>*

Enable DHCP for IPv4. Off by default.

dhcp6: *<(bool)>*

Enable DHCP for IPv6. Off by default.

addresses: *<(sequence of scalars)>*

Add static addresses to the interface in addition to the ones received through DHCP or RA. Each sequence entry is in CIDR notation, i. e. of the form `addr/prefixlen`. `addr` is an IPv4 or IPv6 address as recognized by `inet_pton``(3) and ``prefixlen` the number of bits of the subnet.

Example: `addresses: [192.168.14.2/24, 2001:1::1/64]`

gateway4: or **gateway6**: *<(scalar)>*

Set default gateway for IPv4/6, for manual address configuration. This requires setting `addresses` too. Gateway IPs must be in a form recognized by `inet_pton(3)`

Example for IPv4: `gateway4: 172.16.0.1` Example for IPv6: `gateway6: 2001:4::1`

nameservers: *<(mapping)>*

Set DNS servers and search domains, for manual address configuration. There are two supported fields: `addresses:` is a list of IPv4 or IPv6 addresses similar to `gateway*`, and `search:` is a list of search domains.

Example:

```
nameservers:
  search: [lab, home]
  addresses: [8.8.8.8, FEDC::1]
```

routes: *<(sequence of mapping)>*

Add device specific routes. Each mapping includes a `to`, `via` key with an IPv4 or IPv6 address as value. `metric` is an optional value.

Example:

```
routes:
 - to: 0.0.0.0/0
   via: 10.23.2.1
   metric: 3
```

Ethernets

Ethernet device definitions do not support any specific properties beyond the common ones described above.

Bonds

interfaces *<(sequence of scalars)>*

All devices matching this ID list will be added to the bond.

Example:

```
ethernets:
  switchports:
    match: {name: "enp2*"}
[...]
bonds:
  bond0:
    interfaces: [switchports]
```

parameters: *<(mapping)>*

Customization parameters for special bonding options. Time values are specified in seconds unless otherwise specified.

mode: *<(scalar)>*

Set the bonding mode used for the interfaces. The default is `balance-rr` (round robin). Possible values are `balance-rr`, `active-backup`, `balance-xor`, `broadcast`, `802.3ad`, `balance-tlb`, and `balance-alb`.

lacp-rate: *<(scalar)>*

Set the rate at which LACPDUs are transmitted. This is only useful in 802.3ad mode. Possible values are `slow` (30 seconds, default), and `fast` (every second).

mii-monitor-interval: *<(scalar)>*

Specifies the interval for MII monitoring (verifying if an interface of the bond has carrier). The default is `0`; which disables MII monitoring.

min-links: *<(scalar)>*

The minimum number of links up in a bond to consider the bond interface to be up.

transmit-hash-policy: *<(scalar)>*

Specifies the transmit hash policy for the selection of slaves. This is only useful in balance-xor, 802.3ad and balance-tlb modes. Possible values are `layer2`, `layer3+4`, `layer2+3`, `encap2+3`, and `encap3+4`.

ad-select: *<(scalar)>*

Set the aggregation selection mode. Possible values are `stable`, `bandwidth`, and `count`. This option is only used in 802.3ad mode.

all-slaves-active: *<(bool)>*

If the bond should drop duplicate frames received on inactive ports, set this option to `false`. If they should be delivered, set this option to `true`. The default value is false, and is the desirable behavior in most situations.

arp-interval: *<(scalar)>*

Set the interval value for how frequently ARP link monitoring should happen. The default value is `0`, which disables ARP monitoring.

arp-ip-targets: *<(sequence of scalars)>*

IPs of other hosts on the link which should be sent ARP requests in order to validate that a slave is up. This option is only used when `arp-interval` is set to a value other than `0`. At least one IP address must be given for ARP link monitoring to function. Only IPv4 addresses are supported. You can specify up to 16 IP addresses. The default value is an empty list.

arp-validate: *<(scalar)>*

Configure how ARP replies are to be validated when using ARP link monitoring. Possible values are `none`, `active`, `backup`, and `all`.

arp-all-targets: *<(scalar)>*

Specify whether to use any ARP IP target being up as sufficient for a slave to be considered up; or if all the targets must be up. This is only used for `active-backup` mode when `arp-validate` is enabled. Possible values are `any` and `all`.

up-delay: *<(scalar)>*

Specify the delay before enabling a link once the link is physically up. The default value is `0`.

down-delay: *<(scalar)>*

Specify the delay before disabling a link once the link has been lost. The default value is `0`.

fail-over-mac-policy: *<(scalar)>*

Set whether to set all slaves to the same MAC address when adding them to the bond, or how else the system should handle MAC addresses. The possible values are `none`, `active`, and `follow`.

gratuitious-arp: *<(scalar)>*

Specify how many ARP packets to send after failover. Once a link is up on a new slave, a notification is sent and possibly repeated if this value is set to a number greater than `1`. The default value is `1` and valid values are between `1` and `255`. This only affects `active-backup` mode.

packets-per-slave: *<(scalar)>*

In `balance-rr` mode, specifies the number of packets to transmit on a slave before switching to the next. When this value is set to `0`, slaves are chosen at random. Allowable values are between `0` and `65535`. The default value is `1`. This setting is only used in `balance-rr` mode.

primary-reselect-policy: *<(scalar)>*

Set the reselection policy for the primary slave. On failure of the active slave, the system will use this policy to decide how the new active slave will be chosen and how recovery will be handled. The possible values are `always`, `better`, and `failure`.

learn-packet-interval: *<(scalar)>*

Specify the interval between sending learning packets to each slave. The value range is between `1` and `0x7fffffff`. The default value is `1`. This option only affects `balance-tlb` and `balance-alb` modes.

Bridges

interfaces: *<(sequence of scalars)>*

All devices matching this ID list will be added to the bridge.

Example:

```
ethernets:
  switchports:
    match: {name: "enp2*"}
[...]
bridges:
  br0:
    interfaces: [switchports]
```

parameters: *<(mapping)>*

Customization parameters for special bridging options. Time values are specified in seconds unless otherwise specified.

ageing-time: *<(scalar)>*

Set the period of time to keep a MAC address in the forwarding database after a packet is received.

priority: *<(scalar)>*

Set the priority value for the bridge. This value should be an number between `0` and `65535`. Lower values mean higher priority. The bridge with the higher priority will be elected as the root bridge.

forward-delay: *<(scalar)>*

Specify the period of time the bridge will remain in Listening and Learning states before getting to the Forwarding state. This value should be set in seconds for the systemd backend, and in milliseconds for the NetworkManager backend.

hello-time: *<(scalar)>*

Specify the interval between two hello packets being sent out from the root and designated bridges. Hello packets communicate information about the network topology.

max-age: *<(scalar)>*

Set the maximum age of a hello packet. If the last hello packet is older than that value, the bridge will attempt to become the root bridge.

path-cost: *<(scalar)>*

Set the cost of a path on the bridge. Faster interfaces should have a lower cost. This allows a finer control on the network topology so that the fastest paths are available whenever possible.

stp: *<(bool)>*

Define whether the bridge should use Spanning Tree Protocol. The default value is "true", which means that Spanning Tree should be used.

VLANs

id: *<(scalar)>*

VLAN ID, a number between 0 and 4094.

link: *<(scalar)>*

ID of the underlying device definition on which this VLAN gets created.

Example:

```
ethernets:
  eno1: {...}
vlans:
  en-intra:
    id: 1
    link: eno1
    dhcp4: yes
  en-vpn:
    id: 2
    link: eno1
    address: ...
```

Examples

Configure an ethernet device with networkd, identified by its name, and enable DHCP:

```
network:
  version: 2
  ethernets:
    eno1:
      dhcp4: true
```

This is a complex example which shows most available features:

```
network:
  version: 2
  ethernets:
    # opaque ID for physical interfaces, only referred to by other stanzas
    id0:
      match:
        macaddress: 00:11:22:33:44:55
      wakeonlan: true
      dhcp4: true
      addresses:
        - 192.168.14.2/24
        - 2001:1::1/64
      gateway4: 192.168.14.1
```

```
      gateway6: 2001:1::2
      nameservers:
        search: [foo.local, bar.local]
        addresses: [8.8.8.8]
    lom:
      match:
        driver: ixgbe
      # you are responsible for setting tight enough match rules
      # that only match one device if you use set-name
      set-name: lom1
      dhcp6: true
    switchports:
      # all cards on second PCI bus; unconfigured by themselves, will be added
      # to br0 below
      match:
        name: enp2*
      mtu: 1280
  bonds:
    bond0:
      interfaces: [id0, lom]
  bridges:
    # the key name is the name for virtual (created) interfaces; no match: and
    # set-name: allowed
    br0:
      # IDs of the components; switchports expands into multiple interfaces
      interfaces: [wlp1s0, switchports]
      dhcp4: true
  vlans:
    en-intra:
      id: 1
      link: id0
      dhcp4: yes
  # static routes
  routes:
   - to: 0.0.0.0/0
     via: 11.0.0.1
     metric: 3
```

Network Configuration Outputs

Cloud-init converts various forms of user supplied or automatically generated configuration into an internal network configuration state. From this state Cloud-init delegates rendering of the configuration to Distro supported formats. The following `renderers` are supported in cloud-init:

- **ENI**

/etc/network/interfaces or `ENI` is supported by the `ifupdown` package found in Ubuntu and Debian.

- **Netplan**

Since Ubuntu 16.10, codename Yakkety, the `netplan` project has been an optional network configuration tool which consumes *Networking Config Version 2* input and renders network configuration for supported backends such as `systemd-networkd` and `NetworkManager`.

- **Sysconfig**

Sysconfig format is used by RHEL, CentOS, Fedora and other derivatives.

Network Output Policy

The default policy for selecting a network `renderer` in order of preference is as follows:

- ENI
- Sysconfig
- Netplan

When applying the policy, Cloud-init checks if the current instance has the correct binaries and paths to support the renderer. The first renderer that can be used is selected. Users may override the network renderer policy by supplying an updated configuration in cloud-config.

```
system_info:
  network:
    renderers: ['netplan', 'eni', 'sysconfig']
```

Network Configuration Tools

Cloud-init contains one tool used to test input/output conversion between formats. The `tools/net-convert.py` in the Cloud-init source repository is helpful for examining expected output for a given input format.

CLI Interface :

```
% tools/net-convert.py --help
usage: net-convert.py [-h] --network-data PATH --kind
                      {eni,network_data.json,yaml} -d PATH [-m name,mac]
                      --output-kind {eni,netplan,sysconfig}

optional arguments:
  -h, --help            show this help message and exit
  --network-data PATH, -p PATH
  --kind {eni,network_data.json,yaml}, -k {eni,network_data.json,yaml}
  -d PATH, --directory PATH
                        directory to place output in
  -m name,mac, --mac name,mac
                        interface name to mac mapping
  --output-kind {eni,netplan,sysconfig}, -ok {eni,netplan,sysconfig}
```

Example output converting V2 to sysconfig:

```
% tools/net-convert.py --network-data v2.yaml --kind yaml \
    --output-kind sysconfig -d target
% cat target/etc/sysconfig/network-scripts/ifcfg-eth*
# Created by cloud-init on instance boot automatically, do not edit.
#
BOOTPROTO=static
DEVICE=eth7
IPADDR=192.168.1.5/255.255.255.0
NM_CONTROLLED=no
ONBOOT=yes
TYPE=Ethernet
USERCTL=no
# Created by cloud-init on instance boot automatically, do not edit.
#
BOOTPROTO=dhcp
DEVICE=eth9
```

```
NM_CONTROLLED=no
ONBOOT=yes
TYPE=Ethernet
USERCTL=no
```

Vendor Data

Overview

Vendordata is data provided by the entity that launches an instance (for example, the cloud provider). This data can be used to customize the image to fit into the particular environment it is being run in.

Vendordata follows the same rules as user-data, with the following caveats:

1. Users have ultimate control over vendordata. They can disable its execution or disable handling of specific parts of multipart input.
2. By default it only runs on first boot
3. Vendordata can be disabled by the user. If the use of vendordata is required for the instance to run, then vendordata should not be used.
4. user supplied cloud-config is merged over cloud-config from vendordata.

Users providing cloud-config data can use the '#cloud-config-jsonp' method to more finely control their modifications to the vendor supplied cloud-config. For example, if both vendor and user have provided 'runcmd' then the default merge handler will cause the user's runcmd to override the one provided by the vendor. To append to 'runcmd', the user could better provide multipart input with a cloud-config-jsonp part like:

```
#cloud-config-jsonp
[{ "op": "add", "path": "/runcmd", "value": ["my", "command", "here"]}]
```

Further, we strongly advise vendors to not 'be evil'. By evil, we mean any action that could compromise a system. Since users trust you, please take care to make sure that any vendordata is safe, atomic, idempotent and does not put your users at risk.

Input Formats

cloud-init will download and cache to filesystem any vendor-data that it finds. Vendordata is handled exactly like user-data. That means that the vendor can supply multipart input and have those parts acted on in the same way as user-data.

The only differences are:

- user-scripts are stored in a different location than user-scripts (to avoid namespace collision)
- user can disable part handlers by cloud-config settings. For example, to disable handling of 'part-handlers' in vendor-data, the user could provide user-data like this:

```
#cloud-config
vendordata: {excluded: 'text/part-handler'}
```

Examples

There are examples in the examples subdirectory.

Additionally, the 'tools' directory contains 'write-mime-multipart', which can be used to easily generate mime-multipart files from a list of input files. That data can then be given to an instance.

See 'write-mime-multipart –help' for usage.

Testing and debugging cloud-init

Overview

This topic will discuss general approaches for test and debug of cloud-init on deployed instances.

Boot Time Analysis - cloud-init analyze

Occasionally instances don't appear as performant as we would like and cloud-init packages a simple facility to inspect what operations took cloud-init the longest during boot and setup.

The script **/usr/bin/cloud-init** has an analyze sub-command **analyze** which parses any cloud-init.log file into formatted and sorted events. It allows for detailed analysis of the most costly cloud-init operations are to determine the long-pole in cloud-init configuration and setup. These subcommands default to reading /var/log/cloud-init.log.

- `analyze show` Parse and organize cloud-init.log events by stage and

include each sub-stage granularity with time delta reports.

```
$ cloud-init analyze show -i my-cloud-init.log
-- Boot Record 01 --
The total time elapsed since completing an event is printed after the "@"
character.
The time the event takes is printed after the "+" character.

Starting stage: modules-config
|`->config-emit_upstart ran successfully @05.47600s +00.00100s
|`->config-snap_config ran successfully @05.47700s +00.00100s
|`->config-ssh-import-id ran successfully @05.47800s +00.00200s
|`->config-locale ran successfully @05.48000s +00.00100s
...
```

- `analyze dump` Parse cloud-init.log into event records and return a list of

dictionaries that can be consumed for other reporting needs.

```
$ cloud-init analyze blame -i my-cloud-init.log
[
 {
  "description": "running config modules",
  "event_type": "start",
  "name": "modules-config",
  "origin": "cloudinit",
  "timestamp": 1510807493.0
 },...
```

- `analyze blame` Parse cloud-init.log into event records and sort them based

on highest time cost for quick assessment of areas of cloud-init that may need improvement.

```
$ cloud-init analyze blame -i my-cloud-init.log
-- Boot Record 11 --
     00.01300s (modules-final/config-scripts-per-boot)
     00.00400s (modules-final/config-final-message)
     00.00100s (modules-final/config-rightscale_userdata)
     ...
```

Analyze quickstart - LXC

To quickly obtain a cloud-init log try using lxc on any ubuntu system:

```
$ lxc init ubuntu-daily:xenial x1
$ lxc start x1
# Take lxc's cloud-init.log and pipe it to the analyzer
$ lxc file pull x1/var/log/cloud-init.log - | cloud-init analyze dump -i -
$ lxc file pull x1/var/log/cloud-init.log - | \
python3 -m cloudinit.analyze dump -i -
```

Analyze quickstart - KVM

To quickly analyze a KVM a cloud-init log:

1. Download the current cloud image

   wget https://cloud-images.ubuntu.com/daily/server/xenial/current/xenial-server-cloudimg-amd64.img

2. Create a snapshot image to preserve the original cloud-image

```
$ qemu-img create -b xenial-server-cloudimg-amd64.img -f qcow2 \
test-cloudinit.qcow2
```

3. Create a seed image with metadata using *cloud-localds*

```
$ cat > user-data <<EOF
  #cloud-config
  password: passw0rd
  chpasswd: { expire: False }
  EOF
$  cloud-localds my-seed.img user-data
```

4. Launch your modified VM

```
 $  kvm -m 512 -net nic -net user -redir tcp:2222::22 \
-drive file=test-cloudinit.qcow2,if=virtio,format=qcow2 \
-drive file=my-seed.img,if=virtio,format=raw
```

5. Analyze the boot (blame, dump, show)

```
 $ ssh -p 2222 ubuntu@localhost 'cat /var/log/cloud-init.log' | \
cloud-init analyze blame -i -
```

Running single cloud config modules

This subcommand is not called by the init system. It can be called manually to load the configured datasource and run a single cloud-config module once using the cached userdata and metadata after the instance has booted. Each cloud-config module has a module FREQUENCY configured: PER_INSTANCE, PER_BOOT, PER_ONCE or PER_ALWAYS. When a module is run by cloud-init, it stores a semaphore file in `/var/lib/cloud/instance/sem/config_<module_name>.<frequency>` which marks when the module last successfully ran. Presence of this semaphore file prevents a module from running again if it has already been run. To ensure that a module is run again, the desired frequency can be overridden on the commandline:

```
$ sudo cloud-init single --name cc_ssh --frequency always
...
Generating public/private ed25519 key pair
...
```

Inspect cloud-init.log for output of what operations were performed as a result.

More information

Useful external references

- The beauty of cloudinit
- Introduction to cloud-init (video)

Hacking on cloud-init

This document describes how to contribute changes to cloud-init. It assumes you have a Launchpad account, and refers to your launchpad user as `LP_USER` throughout.

Do these things once

- To contribute, you must sign the Canonical contributor license agreement

  If you have already signed it as an individual, your Launchpad user will be listed in the contributor-agreement-canonical group. Unfortunately there is no easy way to check if an organization or company you are doing work for has signed. If you are unsure or have questions, email Scott Moser or ping smoser in `#cloud-init` channel via freenode.

  When prompted for 'Project contact' or 'Canonical Project Manager' enter 'Scott Moser'.

- Clone the upstream repository on Launchpad:

```
git clone https://git.launchpad.net/cloud-init
cd cloud-init
```

  There is more information on Launchpad as a git hosting site in Launchpad git documentation.

- Create a new remote pointing to your personal Launchpad repository. This is equivalent to 'fork' on github.

```
git remote add LP_USER ssh://LP_USER@git.launchpad.net/~LP_USER/cloud-init
git push LP_USER master
```

Do these things for each feature or bug

- Create a new topic branch for your work:

```
git checkout -b my-topic-branch
```

- Make and commit your changes (note, you can make multiple commits, fixes, more commits.):

```
git commit
```

- Run unit tests and lint/formatting checks with tox:

```
tox
```

- Push your changes to your personal Launchpad repository:

```
git push -u LP_USER my-topic-branch
```

- Use your browser to create a merge request:
  - Open the branch on Launchpad.
    - You can see a web view of your repository and navigate to the branch at:

      `https://code.launchpad.net/~LP_USER/cloud-init/`
    - It will typically be at:

      `https://code.launchpad.net/~LP_USER/cloud-init/+git/cloud-init/+ref/BRANCHNAME`

      for example, here is larsks move-to-git branch: https://code.launchpad.net/~larsks/cloud-init/+git/cloud-init/+ref/feature/move-to-git
  - Click 'Propose for merging'
  - Select 'lp:cloud-init' as the target repository
  - Type '`master`' as the Target reference path
  - Click 'Propose Merge'
  - On the next page, hit 'Set commit message' and type a git combined git style commit message like:

```
Activate the frobnicator.

The frobnicator was previously inactive and now runs by default.
This may save the world some day.  Then, list the bugs you fixed
as footers with syntax as shown here.

The commit message should be one summary line of less than
74 characters followed by a blank line, and then one or more
paragraphs describing the change and why it was needed.

This is the message that will be used on the commit when it
is sqaushed and merged into trunk.

LP: #1
```

Then, someone in the cloud-init-dev group will review your changes and follow up in the merge request.

Feel free to ping and/or join `#cloud-init` on freenode irc if you have any questions.

Integration Testing

Overview

This page describes the execution, development, and architecture of the cloud-init integration tests:

- Execution explains the options available and running of tests
- Development shows how to write test cases
- Architecture explains the internal processes

Execution

Overview

In order to avoid the need for dependencies and ease the setup and configuration users can run the integration tests via tox:

```
$ git clone https://git.launchpad.net/cloud-init
$ cd cloud-init
$ tox -e citest -- -h
```

Everything after the double dash will be passed to the integration tests. Executing tests has several options:

- `run` an alias to run both `collect` and `verify`. The `tree_run` command does the same thing, except uses a deb built from the current working tree.
- `collect` deploys on the specified platform and distro, patches with the requested deb or rpm, and finally collects output of the arbitrary commands. Similarly, `` `tree_collect `` will collect output using a deb built from the current working tree.
- `verify` given a directory of test data, run the Python unit tests on it to generate results.
- `bddeb` will build a deb of the current working tree.

Run

The first example will provide a complete end-to-end run of data collection and verification. There are additional examples below explaining how to run one or the other independently.

```
$ git clone https://git.launchpad.net/cloud-init
$ cd cloud-init
$ tox -e citest -- run --verbose \
    --os-name stretch --os-name xenial \
    --deb cloud-init_0.7.8~my_patch_all.deb \
    --preserve-data --data-dir ~/collection
```

The above command will do the following:

- `run` both collect output and run tests the output
- `--verbose` verbose output
- `--os-name stretch` on the Debian Stretch release
- `--os-name xenial` on the Ubuntu Xenial release

- `--deb cloud-init_0.7.8~patch_all.deb` use this deb as the version of cloud-init to run with
- `--preserve-data` always preserve collected data, do not remove data after successful test run
- `--data-dir ~/collection` write collected data into *~/collection*, rather than using a temporary directory

For a more detailed explanation of each option see below.

Note: By default, data collected by the run command will be written into a temporary directory and deleted after a successful. If you would like to preserve this data, please use the option `--preserve-data`.

Collect

If developing tests it may be necessary to see if cloud-config works as expected and the correct files are pulled down. In this case only a collect can be ran by running:

```
$ tox -e citest -- collect -n xenial --data-dir /tmp/collection
```

The above command will run the collection tests on xenial and place all results into */tmp/collection*.

Verify

When developing tests it is much easier to simply rerun the verify scripts without the more lengthy collect process. This can be done by running:

```
$ tox -e citest -- verify --data-dir /tmp/collection
```

The above command will run the verify scripts on the data discovered in */tmp/collection*.

TreeRun and TreeCollect

If working on a cloud-init feature or resolving a bug, it may be useful to run the current copy of cloud-init in the integration testing environment. The integration testing suite can automatically build a deb based on the current working tree of cloud-init and run the test suite using this deb.

The `tree_run` and `tree_collect` commands take the same arguments as the `run` and `collect` commands. These commands will build a deb and write it into a temporary file, then start the test suite and pass that deb in. To build a deb only, and not run the test suite, the `bddeb` command can be used.

Note that code in the cloud-init working tree that has not been committed when the cloud-init deb is built will still be included. To build a cloud-init deb from or use the `tree_run` command using a copy of cloud-init located in a different directory, use the option `--cloud-init /path/to/cloud-init`.

```
$ tox -e citest -- tree_run --verbose \
    --os-name xenial --os-name stretch \
    --test modules/final_message --test modules/write_files \
    --result /tmp/result.yaml
```

Bddeb

The `bddeb` command can be used to generate a deb file. This is used by the tree_run and tree_collect commands to build a deb of the current working tree. It can also be used a user to generate a deb for use in other situations and avoid needing to have all the build and test dependencies installed locally.

- `--bddeb-args`: arguments to pass through to bddeb
- `--build-os`: distribution to use as build system (default is xenial)
- `--build-platform`: platform to use for build system (default is lxd)
- `--cloud-init`: path to base of cloud-init tree (default is '.')
- `--deb`: path to write output deb to (default is '.')

Setup Image

By default an image that is used will remain unmodified, but certain scenarios may require image modification. For example, many images may use a much older cloud-init. As a result tests looking at newer functionality will fail because a newer version of cloud-init may be required. The following options can be used for further customization:

- `--deb`: install the specified deb into the image
- `--rpm`: install the specified rpm into the image
- `--repo`: enable a repository and upgrade cloud-init afterwards
- `--ppa`: enable a ppa and upgrade cloud-init afterwards
- `--upgrade`: upgrade cloud-init from repos
- `--upgrade-full`: run a full system upgrade
- `--script`: execute a script in the image. This can perform any setup required that is not covered by the other options

Test Case Development

Overview

As a test writer you need to develop a test configuration and a verification file:

- The test configuration specifies a specific cloud-config to be used by cloud-init and a list of arbitrary commands to capture the output of (e.g my_test.yaml)
- The verification file runs tests on the collected output to determine the result of the test (e.g. my_test.py)

The names must match, however the extensions will of course be different, yaml vs py.

Configuration

The test configuration is a YAML file such as *ntp_server.yaml* below:

```
#
# Empty NTP config to setup using defaults
#
# NOTE: this should not require apt feature, use 'which' rather than 'dpkg -l'
# NOTE: this should not require no_ntpdate feature, use 'which' to check for
#       installation rather than 'dpkg -l', as 'grep ntp' matches 'ntpdate'
# NOTE: the verifier should check for any ntp server not 'ubuntu.pool.ntp.org'
cloud_config: |
  #cloud-config
  ntp:
    servers:
```

```
      - pool.ntp.org
required_features:
  - apt
  - no_ntpdate
  - ubuntu_ntp
collect_scripts:
  ntp_installed_servers: |
    #!/bin/bash
    dpkg -l | grep ntp | wc -l
  ntp_conf_dist_servers: |
    #!/bin/bash
    ls /etc/ntp.conf.dist | wc -l
  ntp_conf_servers: |
    #!/bin/bash
    cat /etc/ntp.conf | grep '^server'
```

There are several keys, 1 required and some optional, in the YAML file:

1. The required key is `cloud_config`. This should be a string of valid YAML that is exactly what would normally be placed in a cloud-config file, including the cloud-config header. This essentially sets up the scenario under test.
2. One optional key is `collect_scripts`. This key has one or more sub-keys containing strings of arbitrary commands to execute (e.g. `` `cat /var/log/cloud-config-output.log` ``). In the example above the output of dpkg is captured, grep for ntp, and the number of lines reported. The name of the sub-key is important. The sub-key is used by the verification script to recall the output of the commands ran.
3. The optional `enabled` key enables or disables the test case. By default the test case will be enabled.
4. The optional `required_features` key may be used to specify a list of features flags that an image must have to be able to run the test case. For example, if a test case relies on an image supporting apt, then the config for the test case should include `required_features: [ apt ]`.

Default Collect Scripts

By default the following files will be collected for every test. There is no need to specify these items:

- `/var/log/cloud-init.log`
- `/var/log/cloud-init-output.log`
- `/run/cloud-init/.instance-id`
- `/run/cloud-init/result.json`
- `/run/cloud-init/status.json`
- `` `dpkg-query -W -f='${Version}' cloud-init` ``

Verification

The verification script is a Python file with unit tests like the one, *ntp_server.py*, below:

```
# This file is part of cloud-init. See LICENSE file for license information.

"""cloud-init Integration Test Verify Script"""
from tests.cloud_tests.testcases import base
```

```
class TestNtp(base.CloudTestCase):
    """Test ntp module"""

    def test_ntp_installed(self):
        """Test ntp installed"""
        out = self.get_data_file('ntp_installed_empty')
        self.assertEqual(1, int(out))

    def test_ntp_dist_entries(self):
        """Test dist config file has one entry"""
        out = self.get_data_file('ntp_conf_dist_empty')
        self.assertEqual(1, int(out))

    def test_ntp_entires(self):
        """Test config entries"""
        out = self.get_data_file('ntp_conf_empty')
        self.assertIn('pool 0.ubuntu.pool.ntp.org iburst', out)
        self.assertIn('pool 1.ubuntu.pool.ntp.org iburst', out)
        self.assertIn('pool 2.ubuntu.pool.ntp.org iburst', out)
        self.assertIn('pool 3.ubuntu.pool.ntp.org iburst', out)

# vi: ts=4 expandtab
```

Here is a breakdown of the unit test file:

- The import statement allows access to the output files.
- The class can be named anything, but must import the `base.CloudTestCase`, either directly or via another test class.
- There can be 1 to N number of functions with any name, however only functions starting with `test_*` will be executed.
- There can be 1 to N number of classes in a test module, however only classes inheriting from `base.CloudTestCase` will be loaded.
- Output from the commands can be accessed via `self.get_data_file('key')` where key is the sub-key of `collect_scripts` above.
- The cloud config that the test ran with can be accessed via `self.cloud_config`, or any entry from the cloud config can be accessed via `self.get_config_entry('key')`.
- See the base `CloudTestCase` for additional helper functions.

Layout

Integration tests are located under the *tests/cloud_tests* directory. Test configurations are placed under *configs* and the test verification scripts under *testcases*:

```
cloud-init$ tree -d tests/cloud_tests/
tests/cloud_tests/
- configs
|    - bugs
|    - examples
|    - main
|    - modules
- testcases
    - bugs
    - examples
```

```
    - main
    - modules
```

The sub-folders of bugs, examples, main, and modules help organize the tests. View the README.md in each to understand in more detail each directory.

Test Creation Helper

The integration testing suite has a built in helper to aid in test development. Help can be invoked via `tox -e citest -- create --help`. It can create a template test case config file with user data passed in from the command line, as well as a template test case verifier module.

The following would create a test case named `example` under the `modules` category with the given description, and cloud config data read in from `/tmp/user_data`.

```
$ tox -e citest -- create modules/example \
    -d "a simple example test case" -c "$(< /tmp/user_data)"
```

Development Checklist

- **Configuration File**
  - Named 'your_test.yaml'
  - Contains at least a valid cloud-config
  - Optionally, commands to capture additional output
  - Valid YAML
  - Placed in the appropriate sub-folder in the configs directory
  - Any image features required for the test are specified
- **Verification File**
  - Named 'your_test.py'
  - Valid unit tests validating output collected
  - Passes pylint & pep8 checks
  - Placed in the appropriate sub-folder in the test cases directory
- Tested by running the test:

```
$ tox -e citest -- run -verbose \
    --os-name <release target> \
    --test modules/your_test.yaml \
    [--deb <build of cloud-init>]
```

Architecture

The following section outlines the high-level architecture of the integration process.

Overview

The process flow during a complete end-to-end LXD-backed test.

1. **Configuration**
    - The back end and specific distro releases are verified as supported
    - The test or tests that need to be run are determined either by directory or by individual yaml
2. **Image Creation**
    - Acquire the request LXD image
    - Install the specified cloud-init package
    - Clean the image so that it does not appear to have been booted
    - A snapshot of the image is created and reused by all tests
3. **Configuration**
    - For each test, the cloud-config is injected into a copy of the snapshot and booted
    - The framework waits for `/var/lib/cloud/instance/boot-finished` (up to 120 seconds)
    - All default commands are ran and output collected
    - Any commands the user specified are executed and output collected
4. **Verification**
    - The default commands are checked for any failures, errors, and warnings to validate basic functionality of cloud-init completed successfully
    - The user generated unit tests are then ran validating against the collected output
5. **Results**
    - If any failures were detected the test suite returns a failure
    - Results can be dumped in yaml format to a specified file using the `-r <result_file_name>.yaml` option

Configuring the Test Suite

Most of the behavior of the test suite is configurable through several yaml files. These control the behavior of the test suite's platforms, images, and tests. The main config files for platforms, images and test cases are `platforms.yaml`, `releases.yaml` and `testcases.yaml`.

Config handling

All configurable parts of the test suite use a defaults + overrides system for managing config entries. All base config items are dictionaries.

Merging is done on a key-by-key basis, with all keys in the default and override represented in the final result. If a key exists both in the defaults and the overrides, then the behavior depends on the type of data the key refers to. If it is atomic data or a list, then the overrides will replace the default. If the data is a dictionary then the value will be the result of merging that dictionary from the default config and that dictionary from the overrides.

Merging is done using the function `tests.cloud_tests.config.merge_config`, which can be examined for more detail on config merging behavior.

The following demonstrates merge behavior:

```
defaults:
    list_item:
     - list_entry_1
     - list_entry_2
    int_item_1: 123
    int_item_2: 234
    dict_item:
        subkey_1: 1
        subkey_2: 2
        subkey_dict:
            subsubkey_1: a
            subsubkey_2: b

overrides:
    list_item:
     - overridden_list_entry
    int_item_1: 0
    dict_item:
        subkey_2: false
        subkey_dict:
            subsubkey_2: 'new value'

result:
    list_item:
     - overridden_list_entry
    int_item_1: 0
    int_item_2: 234
    dict_item:
        subkey_1: 1
        subkey_2: false
        subkey_dict:
            subsubkey_1: a
            subsubkey_2: 'new value'
```

Image Config

Image configuration is handled in `releases.yaml`. The image configuration controls how platforms locate and acquire images, how the platforms should interact with the images, how platforms should detect when an image has fully booted, any options that are required to set the image up, and features that the image supports.

Since settings for locating an image and interacting with it differ from platform to platform, there are 4 levels of settings available for images on top of the default image settings. The structure of the image config file is:

```
default_release_config:
    default:
        ...
    <platform>:
        ...
    <platform>:
        ...

releases:
    <release name>:
        <default>:
            ...
```

```
        <platform>:
            ...
        <platform>:
            ...
```

The base config is created from the overall defaults and the overrides for the platform. The overrides are created from the default config for the image and the platform specific overrides for the image.

System Boot

The test suite must be able to test if a system has fully booted and if cloud-init has finished running, so that running collect scripts does not race against the target image booting. This is done using the `system_ready_script` and `cloud_init_ready_script` image config keys.

Each of these keys accepts a small bash test statement as a string that must return 0 or 1. Since this test statement will be added into a larger bash statement it must be a single statement using the `[` test syntax.

The default image config provides a system ready script that works for any systemd based image. If the image is not systemd based, then a different test statement must be provided. The default config also provides a test for whether or not cloud-init has finished which checks for the file `/run/cloud-init/result.json`. This should be sufficient for most systems as writing this file is one of the last things cloud-init does.

The setting `boot_timeout` controls how long, in seconds, the platform should wait for an image to boot. If the system ready script has not indicated that the system is fully booted within this time an error will be raised.

Feature Flags

Not all test cases can work on all images due to features the test case requires not being present on that image. If a test case requires features in an image that are not likely to be present across all distros and platforms that the test suite supports, then the test can be skipped everywhere it is not supported.

Feature flags, which are names for features supported on some images, but not all that may be required by test cases. Configuration for feature flags is provided in `releases.yaml` under the `features` top level key. The features config includes a list of all currently defined feature flags, their meanings, and a list of feature groups.

Feature groups are groups of features that many images have in common. For example, the `Ubuntu_specific` feature group includes features that should be present across most Ubuntu releases, but may or may not be for other distros. Feature groups are specified for an image as a list under the key `feature_groups`.

An image's feature flags are derived from the features groups that that image has and any feature overrides provided. Feature overrides can be specified under the `features` key which accepts a dictionary of `{<feature_name>: true/false}` mappings. If a feature is omitted from an image's feature flags or set to false in the overrides then the test suite will skip any tests that require that feature when using that image.

Feature flags may be overridden at run time using the `--feature-override` command line argument. It accepts a feature flag and value to set in the format `<feature name>=true/false`. Multiple `--feature-override` flags can be used, and will all be applied to all feature flags for images used during a test.

Setup Overrides

If an image requires some of the options for image setup to be used, then it may specify overrides for the command line arguments passed into setup image. These may be specified as a dictionary under the `setup_overrides` key. When an image is set up, the arguments that control how it is set up will be the arguments from the command line, with any entries in `setup_overrides` used to override these arguments.

For example, images that do not come with cloud-init already installed should have `setup_overrides: {upgrade: true}` specified so that in the event that no additional setup options are given, cloud-init will be installed from the image's repos before running tests. Note that if other options such as `--deb` are passed in on the command line, these will still work as expected, since apt's policy for cloud-init would prefer the locally installed deb over an older version from the repos.

Platform Specific Options

There are many platform specific options in image configuration that allow platforms to locate images and that control additional setup that the platform may have to do to make the image usable. For information on how these work, please consult the documentation for that platform in the integration testing suite and the `releases.yaml` file for examples.

Error Handling

The test suite makes an attempt to run as many tests as possible even in the event of some failing so that automated runs collect as much data as possible. In the event that something goes wrong while setting up for or running a test, the test suite will attempt to continue running any tests which have not been affected by the error.

For example, if the test suite was told to run tests on one platform for two releases and an error occurred setting up the first image, all tests for that image would be skipped, and the test suite would continue to set up the second image and run tests on it. Or, if the system does not start properly for one test case out of many to run on that image, that test case will be skipped and the next one will be run.

Note that if any errors occur, the test suite will record the failure and where it occurred in the result data and write it out to the specified result file.

Results

The test suite generates result data that includes how long each stage of the test suite took and which parts were and were not successful. This data is dumped to the log after the collect and verify stages, and may also be written out in yaml format to a file. If part of the setup failed, the traceback for the failure and the error message will be included in the result file. If a test verifier finds a problem with the collected data from a test run, the class, test function and test will be recorded in the result data.

Exit Codes

The test suite counts how many errors occur throughout a run. The exit code after a run is the number of errors that occurred. If the exit code is non-zero then something is wrong either with the test suite, the configuration for an image, a test case, or cloud-init itself.

Note that the exit code does not always directly correspond to the number of failed test cases, since in some cases, a single error during image setup can mean that several test cases are not run. If run is used, then the exit code will be the sum of the number of errors in the collect and verify stages.

Data Dir

When using run, the collected data is written into a temporary directory. In the event that all tests pass, this directory is deleted, but if a test fails or an error occurs, this data will be left in place, and a message will be written to the log giving the location of the data.

Python Module Index

C

Index

C